MASTER EN SCIENCES INFORMATIQUES

Etude des techniques de signature numérique et mise en oeuvre dans le cadre du projet PBFlow

Michot, Olivier

Award date:
1999

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B-5000 NAMUR (BELGIUM)

Etude des techniques de signature numérique et mise en œuvre dans le cadre du projet PBFlow

Olivier Michot

Mémoire présenté en vue de l'obtention du grade de

Licencié en Informatique

Année Académique 1998 - 1999

FACULTÉS UNIVERSITAIRES NOTRE-DAME DE LA PAIX, NAMUR
INSTITUT D'INFORMATIQUE
RUE GRANDGAGNAGE, 21
B-5000 NAMUR (BELGIUM)

Etude des techniques de signature numérique et mise en œuvre dans le cadre du projet PBFlow

Olivier Michot

Résumé

Dans ce mémoire, nous nous sommes intéressés à un des aspects de la Cryptographie : la signature numérique. Il s'agissait d'étudier des techniques de signature numérique et de montrer comment il est possible de les mettre en œuvre dans un cadre concret, le projet `PBFlow`.

Ce document se veut être le guide de l'utilisateur et le livre d'or des outils développés afin de faciliter la maintenance à venir.

Abstract

In this work, we were interested in a specific domain of Cryptography: the digital signature. The goal was to study digital signature techniques and to put them together in order to secures `PBFlow` application.

This document aims to be the user guide and the golden book of the developed tools for all the maintenance procedures to become.

Mémoire présenté en vue
de l'obtention du grade de
Licencié en Informatique
Juin 1999

Promoteur : Prof. J. Ramaekers

Le Général Sun-Tzu disait
« Si tu poses des filets,
tu peux attraper des oiseaux en vol
sans avoir à voler toi-même ».

Table des matières

Remerciements

Tout d'abord, je tiens à remercier le professeur Jean Ramaekers qui m'a proposé ce mémoire et qui m'a guidé tout au long de cette année plus que chargée.

Je remercie également tous les membres de l'équipe `PBFlow` : William Poos, Didier Rossetto et Bernard Ramaekers parce qu'ils ont su m'intégrer au sein de leur groupe et parce qu'ils n'ont pas cesser de m'éclairer de leur lanterne. Grâce à eux, j'ai toujours eu à ma disposition les informations et les outils de travail nécessaires à mes recherches mais j'ai également pu évoluer dans une bonne ambiance, ce qui m'importe beaucoup.

Si je remets ce mémoire dans les temps, c'est grâce à toutes les personnes qui m'ont soutenu et qui ont cru en moi du début jusqu'à la fin. Si je ne cite pas de noms, c'est par crainte d'en oublier, tellement ils sont nombreux.

Merci à toute la famille Deharre et tout particulièrement à Catherine, ma future épouse, à qui je dédie ce travail du fond de mon cœur.

Introduction

Si la signature manuscrite a su, depuis toujours, garantir l'authenticité des documents au bas desquels elle était apposée, dans notre monde moderne de communications au travers des réseaux informatiques, elle est devenue totalement obsolète.

Il a donc fallu trouver une nouvelle solution. Elle nous vient des Mathématiques et plus particulièrement de la Cryptographie qui permet à un auteur d'associer à ses messages, un nombre dont tout le monde peut vérifier l'authenticité mais que lui seul peut avoir généré. La signature numérique est née et comme technique d'authentification, elle est promise à un bel avenir.

Une architecture concrète permettant l'emploi de la signature numérique n'est pas une mince affaire à mettre en place dans la mesure où celle-ci est basée sur la connaissance d'un seul et unique secret : une clé dont la confidentialité et l'intégrité devront être garanties à tous moments. Cela implique une action à grande échelle recouvrant plusieurs spécialités telles que les Mathématiques, l'Electronique et bien sûr l'Informatique.

Dans ce mémoire de licence, nous nous intéresserons aux techniques de signature numérique en tant qu'outils de programmation dans le but de développer, dans le cadre du projet `PBFlow`, un module autonome de signature numérique.

En effet, le projet `PBFlow` et l'application qu'il implémente gèrent électroniquement les échanges de documents inhérents à la gestion d'un permis d'urbanisme mais aucune solution de signature numérique n'a encore été envisagée. Notre objectif sera d'étudier et de mettre en œuvre des techniques de signature numérique adaptées au projet `PBFlow` mais néanmoins génériques.

Nous travaillerons en deux parties :

La première introduira ou rappellera des notions théoriques telles que la fiabilité des systèmes, la cryptographie à clé secrète et à clé privée et les protocoles de base de la sécurité informatique.

La seconde partie pourra être perçue comme le film de notre projet de développement : nous y ferons apparaître les phases de préparation, de développement et de déploiement. La phase de maintenance, quant à elle, sera laissée aux bons soins de l'équipe `PBFlow`.

Le fait que nous ne prenons pas en main la maintenance et que nous n'avons pas produit de documents techniques en cours de développement, nous a conduit à structurer ce travail de telle sorte qu'il puisse être utilisé comme le guide de référence pour des éventuelles améliorations.

PARTIE I : Aspects théoriques

Nous avons choisi de faire précéder la partie pratique de ce travail par une rapide étude des aspects théoriques sur lesquels notre réalisation s'est basée. De la sorte, le lecteur pourra trouver toutes les notions théoriques dont il a besoin pour mieux appréhender les mécanismes mis en place dans le cadre de ce mémoire.

Chapitre I : Introduction à la sécurité informatique

Dans ce chapitre, nous introduirons quelques-unes des notions fondamentales de la sécurité informatique : après avoir défini ce que nous entendons par sécurité informatique, nous montrerons que celle-ci présente une nature particulière. Il s'agira également de décrire les objectifs qu'elle vise à atteindre et de mettre en évidence le domaine d'application de ce mémoire.

1. *Définition*

S'il est apparemment facile de protéger ses informations stratégiques en les enfermant dans un coffre ou d'assurer le transport d'un document confidentiel en le faisant parvenir à son destinataire sous pli scellé, si la signature manuscrite apposée en bas d'une feuille suffit à confirmer la validité d'un document, l'utilisation quotidienne des ordinateurs et des télécommunications pose le problème de la sécurité sous un nouveau jour : la *sécurité informatique*.

Ce nouveau type de sécurité présente un aspect global. La figure 1.1. schématise la sécurité informatique comme l'interaction entre des êtres humains, des systèmes informatiques et un environnement souvent hostile.

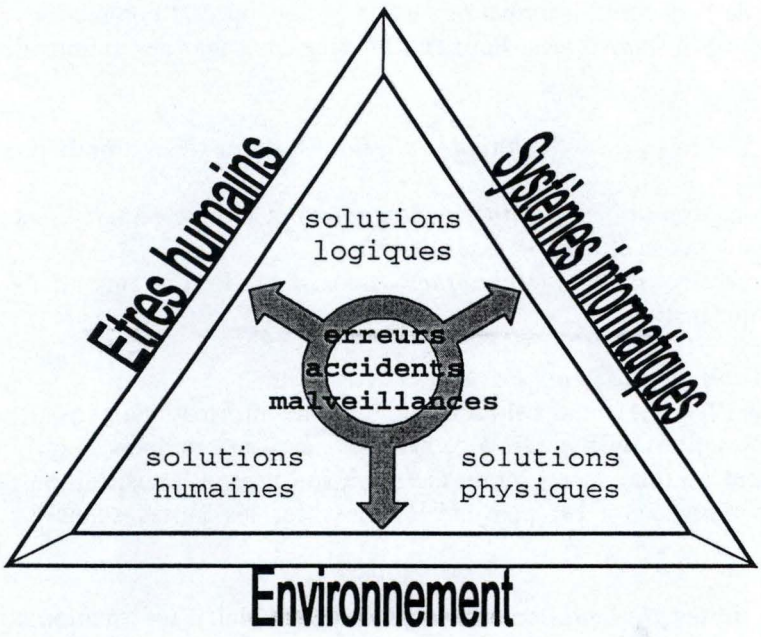

Figure 1.1. : aspect global de la sécurité informatique.

Afin de faire apparaître au mieux le fait que la sécurité informatique est un problème global et qu'elle ne sera garantie que si on la considère comme telle, nous proposons la définition suivante :

« La sécurité informatique est la gestion des techniques physiques, logiques et humaines destinées à protéger, contre les accidents, les erreurs et les comportements malicieux, les êtres humains qui confient certaines valeurs à des systèmes informatiques fonctionnant dans un certain environnement. »

Cette définition mérite quelques explications …

… gestion des techniques …
Il s'agira de choisir les techniques adéquates pour atteindre au mieux un ensemble d'objectifs fixés préalablement. Bien souvent, les organisations se fixent un niveau minimal de sécurité à atteindre (un niveau de risque acceptable) et décide des moyens qu'elles sont prêtes à mettre en œuvre pour atteindre ces objectifs.
On peut classer les différentes catégories de techniques selon qu'elles soient :

1. ***physiques*** : emploi de moyens matériels (extincteurs, portes coupe-feu ; …),
2. ***logiques*** : mise en place de solutions logiques (firewalls, cryptographie, …),
3. ***humaines*** : mise en œuvre de moyens humains (contrat d'assurance, …).

… contre les accidents, les erreurs et les comportements malicieux …
Les menaces qui pèsent sur un système d'information appartiennent à trois grandes catégories :

1. ***les accidents***,
2. ***les erreurs*** produites sans intention de nuire au système,
3. ***Les malveillances*** qui dénotent d'un caractère intentionnel.

… destinées à protéger … les êtres humains …
In fine, le but de la sécurité informatique est la protection de l'homme lorsqu'il utilise un ou plusieurs systèmes informatiques. Pour cela, il faut envisager des actions de sécurisation sur trois niveaux :

1. protection directe (***techniques physiques***) du matériel afin de protéger l'homme et ses données,
2. protection directe (***techniques logiques***) et indirecte (par protection du matériel) des données afin de protéger l'homme,
3. protection directe (***techniques humaines***) de l'homme dans son emploi de l'informatique.

… qui confient certaines de leurs valeurs …
Les risques que l'homme prend s'il utilise un système informatique peuvent se mesurer par la valeur des informations qu'il confie à ce système : ce sont ces données qu'il risque de perdre (momentanément ou totalement) ou de mettre, à son insu, à la disposition de personnes non autorisées. De nouveau, il est possible de classifier les pertes selon les trois catégories suivantes :

1. ***les pertes physiques*** comme la perte de matériel, d'information ou d'argent du fait de la non utilisation du système,
2. ***les pertes logiques*** comme la perte de confiance, la perte de clientèle, les pertes dues aux frais supplémentaires occasionnés,
3. ***Les pertes humaines*** comme les préjudices corporels ou moraux.

… des systèmes informatiques dans un environnement …
Un élément capital à prendre en compte pour la sécurité informatique est l'*environnement*. Il faudra tenir compte de l'environnement physique (la météo, les catastrophes naturelles, …), de

l'environnement logique (les nouvelles techniques, la concurrence, ...) et de l'environnement humain (la situation politique et sociale, ...).

2. *Nature particulière du problème*

La sécurité informatique présente une nature particulière. En effet, il s'agit autant de protéger un système d'information que des biens matériels. Les biens matériels (disques, bandes magnétiques, ...) ne sont que des moyens grâce auxquels l'entreprise fait de l'*informatique*, c'est-à-dire du traitement d'informations. L'information est la richesse et le cœur de l'organisation. C'est pourquoi la sécurité informatique prend cet aspect particulier : il ne s'agit plus seulement de protéger du matériel, il faut envisager la protection des données informatiques. La protection de ces données impose une action d'ensemble au niveau du matériel, des logiciels de traitement et du personnel dans le cadre spécifique de l'organisation et de son environnement extérieur le plus large.

Notons qu'aucun système de sécurité n'offre une protection totale : si certains risques sont supprimés ou diminués, d'autres subsistent. Les investissements à consentir pour la prévention sont donc à étudier en fonction du niveau minimum de risque que l'on s'est fixé. Il restera encore à envisager des mesures de reprise en cas d'accidents inévitables.

3. *Objectifs de la sécurité informatique*

Nous avons proposé, dans notre définition, un objectif ultime à la sécurité informatique : la protection de l'homme au travers des valeurs qu'il confie à un système informatique. Nous avons également indiqué que cette protection devait s'envisager globalement.

Nous diviserons l'objectif de la sécurité en trois sous-buts : la protection du matériel, la protection des données et la protection du personnel.

A chacun de ces sous-buts, nous associons trois qualités fondamentales qu'il faudra garantir :

- **la disponibilité**
- **l'intégrité**
- **la confidentialité**

3.1. Disponibilité

Un bon système informatique doit pouvoir garantir que le matériel et les données seront toujours accessibles. Il s'agit d'un problème de ***disponibilité***. Les données et le matériel ne doivent être disponibles que pour les personnes autorisées. On parle dans ce cas d'***accessibilité***. Les accès effectués doivent être mémorisés afin de gérer les quotas ou la facturation des services. De plus, il faut pouvoir disposer de son matériel, de ses données et du personnel aussi longtemps qu'on le désire. Nous parlerons ainsi de ***pérennité***. Il faut également veiller à ce que disponibilité rime avec ***performance*** afin que les ressources soient accessibles dans des délais raisonnables. Nous parlerons encore d'***audit*** (de sécurité) et d'auditabilité dans le cas où nous évaluons l'efficacité des mesures de sécurisation prises.

3.2. Intégrité

L'***intégrité*** d'une donnée (ou d'un matériel) est sa propriété à être conservée intacte sans dommage et/ou sans perte, et de n'être détruite ou transformée que par l'intervention de personnes autorisées. L'***intégrité*** d'une donnée (ou d'un matériel) peut être évaluée au travers de deux qualités que sont l'***authenticité*** de son contenu et l'***authenticité*** de son origine. Lorsque ces deux qualités sont réunies, on est assuré que le contenu est bien ce qu'il doit être et qu'il provient bien de la personne autorisée à le produire.

3.3. Confidentialité

La ***confidentialité*** assure que le sens attaché à une information ne sera compris que par des personnes habilitées. Cette information peut être représentée par des données, par du matériel (via les données qu'il contient) mais également par le personnel et ses connaissances. La ***confidentialité*** est donc la protection du secret.

3.4. Fiabilité

Les trois qualités que nous venons de présenter peuvent être mesurées séparément. Cependant, en pratique, elles sont très fortement liées et de ce fait, difficilement dissociables. Elles se recoupent souvent et les mécanismes qui leur assurent un niveau respectable, sont généralement valables pour plusieurs d'entre elles.

Lorsque l'on dispose d'un système informatique qui respecte raisonnablement la *confidentialité*, la *disponibilité* et l'*intégrité*, le système sera considéré comme ***fiable***. Plus un système est fiable, plus les valeurs que l'homme lui confie seront en sécurité. La figure 1.2. résume les qualités essentielles d'un système informatique.

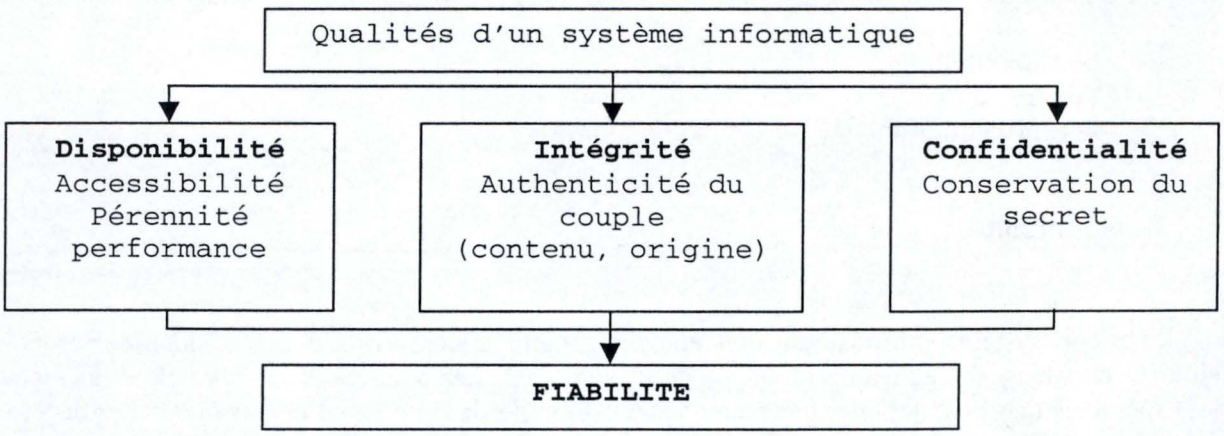

Figure 1.2. : qualités d'un système informatique.

Les erreurs, les accidents et les malveillances ont pour effet de diminuer les niveaux de *confidentialité*, de *disponibilité* et d'*intégrité* du système informatique et donc de diminuer de la même manière son degré de ***fiabilité***. Le but de la sécurité informatique est d'assurer la fiabilité du système.

4. *Domaine d'application*

La sécurité informatique est un problème global, nous l'avons vu et les mesures de protection des systèmes doivent être prises dans plusieurs domaines (matériel, logique et humain) et à tous les niveaux de l'organisation.

Dans ce mémoire, il ne s'agira pas de traiter la sécurité informatique dans son ensemble mais bien de se concentrer sur une des facettes de cette problématique : la sécurité logique. Plus précisément encore, notre but sera de mettre en place un protocole de signature numérique dans le cadre d'échanges de documents pour lesquels l'intégrité du contenu et l'authenticité de l'origine doivent être garantis.

Dans les deux chapitres suivants, nous étudierons respectivement la cryptographie et quelques-uns des protocoles de base afin que chacun puisse mieux comprendre les mécanismes utilisés dans la seconde partie de ce mémoire consacrée aux réalisations pratiques.

Chapitre II : cryptographie

Le mot cryptographie vient des mots grecs χρυπτο (caché ou secret) et γραφη (écriture) : elle signifie l'art de l'écriture secrète. La cryptographie consiste à faire subir des transformations de chiffrement à un texte au moyen de jeu de clés, de manière à le rendre illisible par celui qui ne possède pas le secret et à le restituer en clair par des transformations de déchiffrement.

Dans ce chapitre, nous décrirons les notions et les mécanismes fondamentaux de chacune des trois catégories de cryptographies. Il s'agit respectivement de la ***cryptographie à clé secrète,*** de la ***cryptographie à clés privées*** et des ***fonctions de hash.***

Dans le chapitre suivant, nous montrerons comment ces techniques de base peuvent être rassemblées dans des protocoles en vue d'atteindre les objectifs de sécurité informatique présentés au premier chapitre.

1. *Définitions*

Les menaces qui pèsent sur les données d'un système informatique sont nombreuses. Il est impossible de dresser une liste exhaustive de toutes les attaques possibles. Toutefois, deux catégories d'attaques apparaissent immédiatement :

- Les ***attaques passives*** où le fraudeur se contente d'observer l'information sans la modifier.
- Les ***attaques actives*** où le fraudeur modifie l'information.

Notons que sous le terme de *modification* se cachent de nombreuses variantes. En effet, si l'on considère, par exemple, un courrier électronique, on peut envisager :

- de modifier son ***contenu*** c.-à-d. de changer son sens,
- de modifier l'**auteur** et/ou le **destinataire**,
- de **dupliquer** le message,
- de **subtiliser** le message.

Nous pourrions également envisager le cas où un attaquant introduit de faux messages.

Introduisons les termes suivants :

- Un ***cryptosystème*** est un procédé mathématique pour coder ou transformer d'une façon unique un message écrit en clair en un message dit chiffré, afin qu'il soit inintelligible pour ceux à qui il n'est pas destiné.
- Le ***cryptogramme*** est le texte écrit en caractères secrets.
- Le ***chiffrement*** consiste à transformer un texte clair en un cryptogramme.
- Le ***cryptanalyse*** est l'opération qui consiste à traduire des messages chiffrés pour lesquels on ne possède pas la clé.
- La ***cryptologie*** est la science du chiffre dans ses deux aspects : la cryptographie et la cryptanalyse.

2. *Cryptographie à clé secrète*

La cryptographie à clé secrète utilise une seule clé. Etant donné un message (*texte en clair*) et une clé, le chiffrement produit un texte inintelligible (*cryptogramme*), qui a la même longueur que le texte en clair. Le déchiffrement est l'opération inverse du chiffrement et utilise la même clé que celui-ci (figure 2.1.).

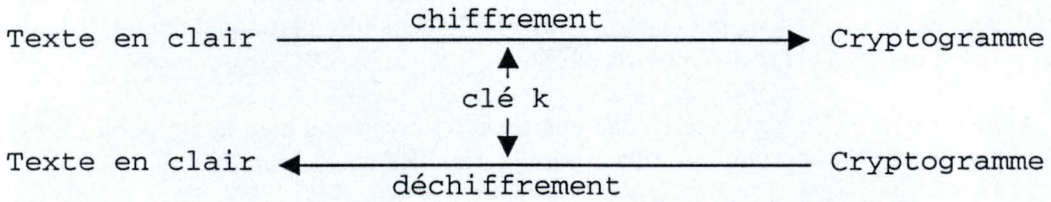

Figure 2.1. : cryptographie à clé secrète.

Il est souvent impossible d'empêcher l'écoute frauduleuse lorsqu'on transfert des informations sur un canal non sécurisé. Si nous nous mettons d'accord pour partager un secret (la clé), en utilisant la cryptographie à clé secrète, nous pouvons nous envoyer des messages l'un à l'autre sur un canal qui peut être écouté, sans nous préoccuper de cette écoute : il suffit à l'auteur de chiffrer son message et au destinataire de déchiffrer le message reçu (en utilisant la clé secrète). Un fraudeur écoutant la ligne n'entendrait que des données inintelligibles. C'est l'emploi courant de la cryptographie que résume la figure 2.2.

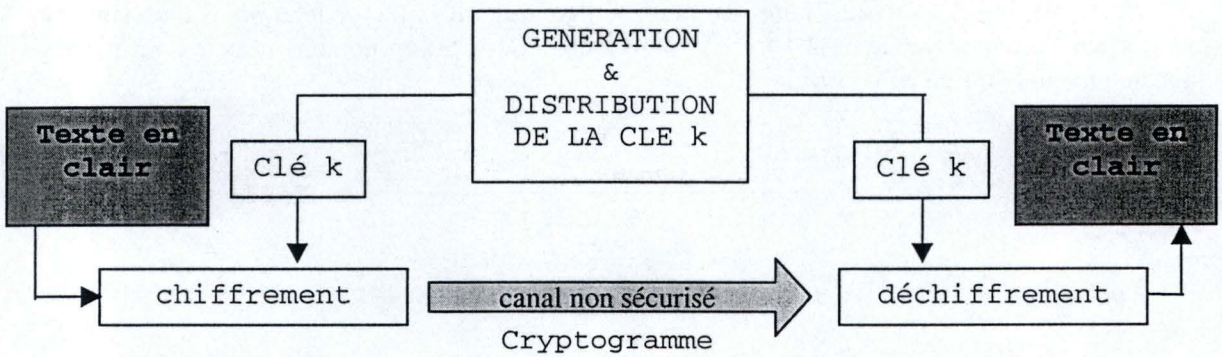

Figure 2.2. : utilisation de la clé secrète.

Nous l'avons dit, les deux entités qui dialoguent partagent un secret : une clé secrète k commune. Les algorithmes de chiffrement et de déchiffrement , notés respectivement E (Encryption) et D (Decryption), peuvent être connus de tous – il s'agit d'algorithmes publics. Par contre, les opérations de chiffrement et de déchiffrement, notés respectivement E_k et D_k, ne sont réalisables que par ceux qui détiennent le secret k puisque par définition, il est impossible d'appliquer ces opérations sans connaître la clé.

Etant donné M, un texte en clair, et M', le cryptogramme correspondant, nous pouvons introduire les notations suivantes :

$$M' = E_k(M) = E(k,M) = \{M\}^k$$
$$M = D_k(M') = D(k,M)$$

3. *Cryptographie à clé publique*

La cryptographie à clé publique est aussi connue sous le nom de cryptographie asymétrique.

A l'opposé de la cryptographie à clé secrète, les clés ne sont pas partagées. Au lieu de cela, chaque individu possède deux clés : une clé privée qui ne peut, en aucun cas, être divulguée et une clé publique qui de préférence, doit être connue de tous.

Notons que nous employons le vocable *privée* et non *secrète* pour définir la clé qui ne peut être divulguée. De la sorte, nous pouvons aisément faire, dans n'importe quel contexte, la distinction entre les cryptographies à clé secrète et à clé publique.

A chacune des clés est associée une opération de cryptographie : la clé publique est utilisée pour le chiffrement et la clé privée est utilisée pour le déchiffrement (figure 2.3.).

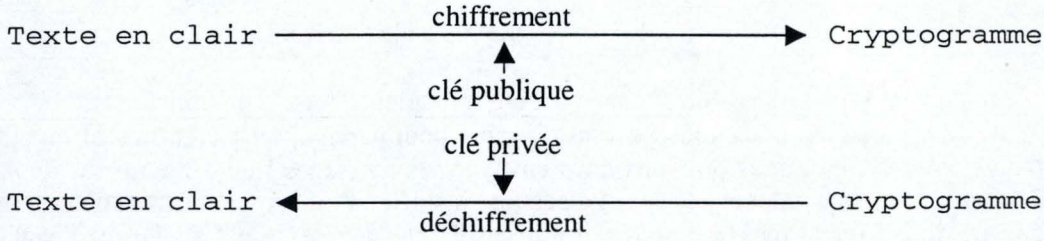

Figure 2.3. : cryptographie à clé publique.

Toutefois, nous pouvons faire un autre usage des clés : la génération d'une signature numérique sur un message (figure 2.4.). La signature numérique est un nombre associé à un message de la même manière qu'un `checksum`[1].

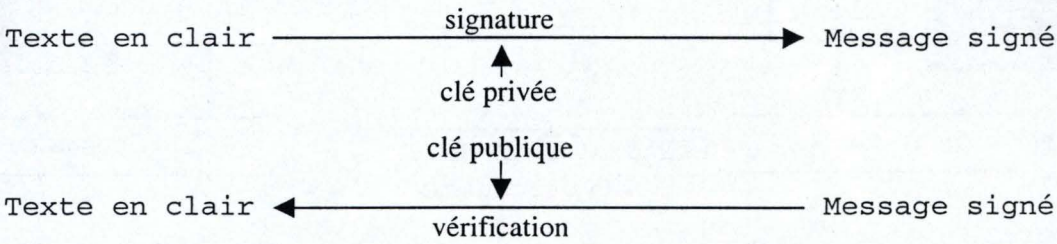

Figure 2.4. : signature numérique.

Supposons qu'Alice soit la personne qui veuille signer un document et que Bob en soit le destinataire[2]. Tout le monde (Bob en particulier) peut connaître la clé publique d'Alice pk_A mais seule Alice connaît et garde précieusement sa clé privée que nous noterons sk_A par opposition à pk_A.

Les deux clés sont liées l'une à l'autre de façon telle que tout message chiffré avec pk_A ne puisse être déchiffré qu'à l'aide de sk_A.

---

[1] A l'instar des bits de parité, le `checksum` est un nombre dont la valeur dépend directement du message dont il assure l'authenticité du contenu. Au chapitre suivant, nous verrons que la signature numérique permet de garantir à la fois l'authenticité de l'origine et du contenu du message sur lequel elle est apposée.

[2] Dans toute la suite de ce travail, nous supposerons qu'Alice et Bob sont les acteurs privilégiés des échanges de documents.

La figure 2.5. schématise la génération et le partage de la paire de clés.

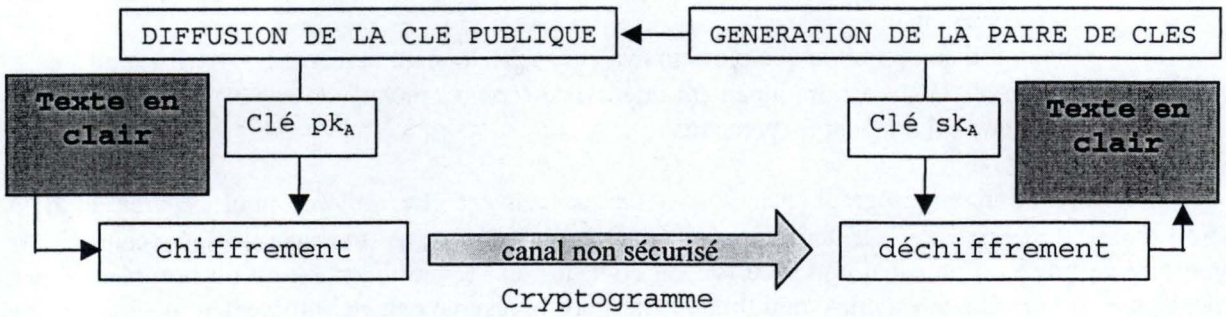

Figure 2.5. : génération de la paire de clés et diffusion de la clé publique.

Comme les cryptosystèmes symétriques (à clé secrète), les algorithmes de chiffrement et de déchiffrement, notés respectivement E et D, peuvent être connus de tous. Dans notre cas, l'opération de chiffrement E_{pkA} est réalisable par tous. Par contre, le déchiffrement D_{skA}, ne peut être appliqué que par le propriétaire de sk_A.

Nous avons donc :

$M' = E_{pkA}(M)$
$M = D_{skA}(M')$

Nous avons vu que nous pouvions générer une signature numérique sur un message en utilisant la clé privée sk_A et que nous pouvions valider une signature en utilisant la clé publique, pk_A. Nous écrirons dans ce cas :

$M'' = D_{skA}(M) = S_{skA}(M)$ pour signature
$M = E_{pkA}(M'') = V_{pkA}(M'')$ pour vérification

Autrement dit, la clé privée sk_A peut être utilisée pour déchiffrer (D_{skA}) des messages que tout le monde a pu chiffrer et pour générer une signature (S_{skA}) que tout le monde peut vérifier avec la clé publique pk_A.

Nous pouvons noter la propriété d'inversion suivante :

$D_{skA}(E_{pkA}(M)) = M = V_{pkA}(S_{skA}(M)) = E_{pkA}(D_{skA}(M))$

4. *Fonctions de hash*

Les fonctions de hash sont mieux connues sous les noms de fonctions à sens unique (one-way function) ou de résumé (message digest). Une fonction cryptographique à sens unique est une transformation mathématique qui prend un message de longueur variable et qui rend un nombre de longueur fixe dépendant directement du contenu du message passé en argument.

Nous noterons le résumé du message M par h(M). Ce résumé a les propriétés suivantes :

- Pour tout M, il est facile de calculer h(M).

- Etant donné `h(M)`, il n'existe aucun moyen de trouver un `M` qui corresponde à `h(M)`, plus facile ou plus rapide que le test de tous les `M` possibles et le calcul de leur `h(M)` correspondant.
- Bien qu'il soit possible que plusieurs valeurs de `M` donnent le même `h(M)` (il y a plus de `M` que de `h(M)`), il est pratiquement impossible (*computationally infeasible*) de trouver deux messages qui donnent le même résumé.

Les fonctions cryptographiques à sens unique peuvent être utilisées pour générer un `MIC` (`Message Integrity Code`) afin de garantir l'intégrité d'un message lorsque celui-ci est transféré au travers d'un canal non sécurisé. En effet, il n'existe pratiquement qu'un seul résumé par message et si le message est corrompu lors du transfert, le résumé calculé initialement ne correspond plus.

Cependant, remarquons que si on envoie « simplement » le message et son résumé, cela ne protège pas entièrement l'intégrité puisqu'un fraudeur pourrait modifier le message et recalculer le résumé correspondant.

Nous verrons au chapitre suivant comment les résumés peuvent être combinés avec d'autres outils de cryptographie tels que la signature numérique.

Chapitre III : protocoles de base

Dans ce chapitre, nous présentons les protocoles élémentaires de la sécurité informatique. Il s'agit des protocoles de chiffrement, d'authentification et de signature numérique. Nous nous intéresserons également au protocole de gestion des clés, car il introduit des notions comme autorité de certification et certificat numérique.

Au travers de ces protocoles, nous verrons comment les techniques cryptographiques exposées au chapitre précédent, peuvent être utilisées pour garantir la confidentialité, l'authenticité du contenu et l'authenticité de l'origine des données qu'elles protègent.

Pour faciliter la lecture et la compréhension des protocoles que nous exposerons dans ce chapitre, nous introduisons les notations suivantes :

- k_{AB} est la clé secrète partagée par A et B,
- pk_X est la clé publique de X,
- sk_X est la clé privée de X,
- sk_{AB} est la clé de session partagée par A et B,
- M est le message transmis ou stocké.

Notons que les lettres A, pour Alice, et B, pour Bob, représentent aussi bien des machines que des personnes. Alice sera toujours l'initiatrice de l'échange.

1. *Protocoles de chiffrement*

Les protocoles de chiffrement définissent l'emploi des algorithmes de chiffrement afin de garantir la **confidentialité** des messages sur lesquels ils sont appliqués.

1.1. Chiffrement à clé secrète

Nous avons vu que dans le cas de la cryptographie à clé secrète, il n'y a qu'une seule clé – la clé secrète partagée par les deux intervenants. Si Alice veut envoyer un message M à Bob et si elle veut être certaine que seul Bob aura l'opportunité de comprendre le sens de ce message, il lui faut chiffrer M à l'aide de la clé secrète k_{AB}. La figure 3.1. présente le protocole de chiffrement à clé secrète.

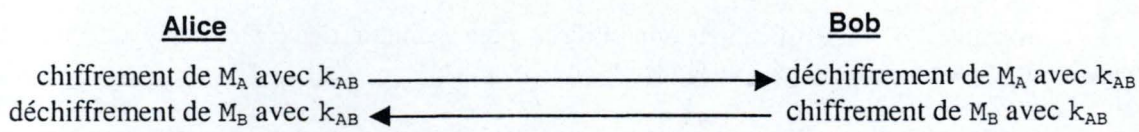

Figure 3.1. : chiffrement à clé secrète.

1.2. Chiffrement à clé publique

Supposons un instant que la paire (clé publique, clé privée) d'Alice soit (pk_A, sk_A) et que la paire (clé publique, clé privée) de Bob soit (pk_B, sk_B). Supposons encore qu'Alice connaisse la clé publique de Bob et que ce dernier connaisse la clé publique d'Alice. Si Alice veut envoyer un message à Bob de manière à ce que lui seul puisse le comprendre, il lui suffit de chiffrer le message à l'aide de la clé publique de Bob (figure 3.2.).

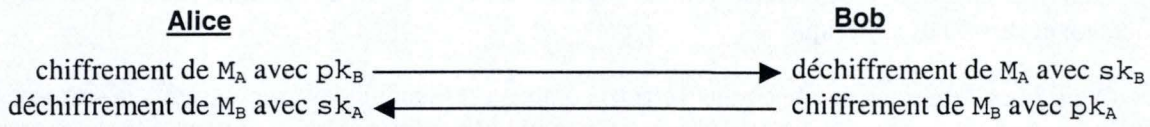

Figure 3.2. : chiffrement à clé publique.

La cryptographie à clé publique peut remplir les mêmes fonctions que la cryptographie à clé secrète si ce n'est que les algorithmes asymétriques sont plus lents que leurs homologues symétriques. On préférera donc utiliser la cryptographie à clé publique là où la cryptographie à clé secrète ne peut fonctionner. C'est pourquoi, la plupart des protocoles de sécurité des réseaux informatiques font appel aux deux techniques : par exemple, la cryptographie à clé publique sert, au début d'une connexion, pour l'établissement d'une clé temporaire de session sk_{AB} – une clé secrète (cryptographie symétrique) utilisée exclusivement pour la nouvelle connexion. Cette clé de session sera utilisée pour le chiffrement et le déchiffrement du reste de la communication.

Supposons qu'Alice veuille parler à Bob. Elle utilise la clé publique pk_B pour chiffrer une clé de session (la clé secrète sk_{AB}) et la clé de session pour chiffrer toutes les données qu'elle veut transmettre. Elle transmet alors, à Bob, la clé de session chiffrée et le message chiffré. Seul Bob peut déchiffrer la clé de session – puisqu'il est le seul propriétaire de sa clé privée sk_B. Il peut alors communiquer, en utilisant la clé de session, avec Alice. La figure 3.3. représente le protocole de chiffrement avec clé de session.

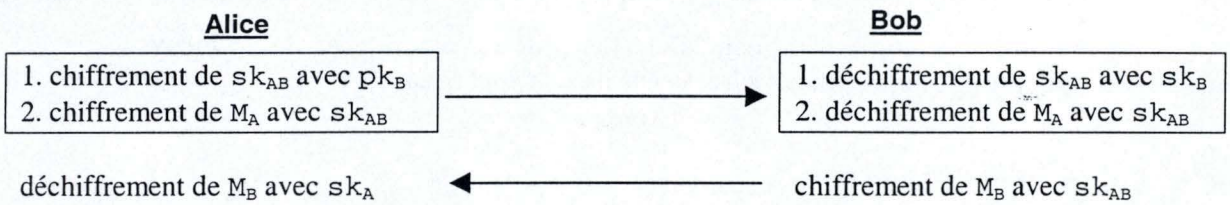

Figure 3.3. : chiffrement avec clé de session.

2. *Protocoles d'authentification*

Les protocoles d'authentification sont utilisés pour garantir deux propriétés différentes : l'authenticité d'acteur et l'authenticité du contenu d'un document.

- L'**authentification d'acteur** consiste à prouver l'identité de chacun des acteurs mis en jeu lors d'une communication.
- L'**authentification du contenu** est le mécanisme qui permet de s'assurer qu'un document n'a pas été corrompu lors de son transfert ou pendant son stockage.

2.1. Authentification d'acteur et cryptographie à clé secrète

Le terme ***authentification forte*** traduit le fait que quelqu'un puisse prouver la connaissance d'un secret sans devoir le révéler. Cette authentification forte est possible avec la cryptographie et elle est particulièrement utile lorsque deux ordinateurs tentent de communiquer au travers d'un canal non sécurisé.

Supposons qu'Alice et Bob, partageant une clé k_{AB}, décident de vérifier qu'ils discutent bien l'un avec l'autre. Chacun d'eux choisi un nombre aléatoire, connu sous le nom de ***challenge***. Alice choisit r_A et Bob choisit r_B. La valeur x chiffré avec la clé k_{AB} est nommée la ***réponse*** au ***challenge*** x (figure 3.4.).

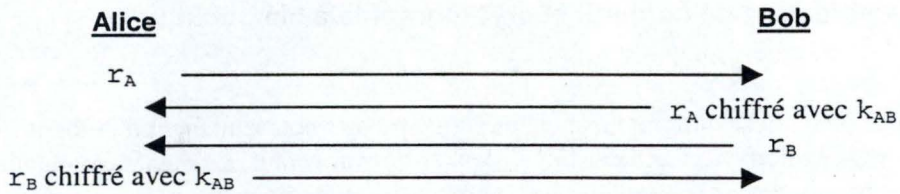

Figure 3.4. : authentification par chiffrement symétrique.

2.2. Authentification d'acteur et cryptographie à clé publique

L'authentification d'acteur est un des domaines où la cryptographie à clé publique apporte le plus. Dans le cas de la cryptographie à clé secrète, si Alice et Bob veulent communiquer, ils doivent partager le même secret. Si Bob veut être capable de prouver son identité à plusieurs personnes, il lui faut connaître un grand nombre de clés (une clé différente pour chaque personne).

La cryptographie à clé publique est plus sûre : en effet, Alice ne doit conserver en mémoire qu'une seule clé, sa clé privée. Si elle veut prouver son identité, il lui suffit de signer un *challenge* avec sa clé privée. Si elle souhaite vérifier des identités, elle devra connaître les clés publiques de chacun de ses interlocuteurs. Cependant, notons que les entités qui vérifient les identités sont souvent des ordinateurs (qui ne craignent pas de retenir des milliers de clés) et que les entités qui prouvent leur identité sont généralement des personnes (qui ne désirent pas devoir retenir beaucoup d'informations).

La figure 3.5. présente un exemple de protocole d'authentification utilisant la cryptographie à clé publique dans le cas où chacun des intervenants connaît la clé publique de l'autre : Alice prouve ici son identité de façon irréfutable.

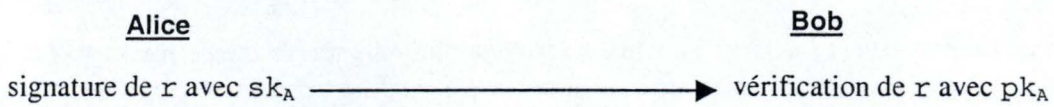

Figure 3.5. : authentification par chiffrement asymétrique.

2.3. Authentification de contenu et cryptographie à clé secrète

Nous pouvons utiliser la cryptographie à clé secrète pour générer un `checksum` cryptographique sur un message et protéger celui-ci contre des modifications accidentelles ou

intentionnelles. Le terme de checksum est dérivé de l'opération de calcul de la somme des blocs de taille fixe constituant le message. Le destinataire du message recalcule la somme et la compare avec le checksum envoyé. Si le message a été manipulé en cours de route, les deux sommes ne correspondent plus et le message doit être considéré comme non valide. Cependant, un fraudeur qui modifie le message peut créer lui-même un nouveau checksum étant donné que les algorithmes de calcul sont publics.

Pour protéger un message contre ces changements malicieux, un checksum ***secret*** (chiffré avec la clé secrète) doit être ajouté lors de l'échange, il constitue le code d'intégrité du message (MIC). Si quelqu'un modifie le message sans connaître la clé, il lui faudra deviner le bon MIC correspondant au nouveau message, ce qui est pratiquement impossible.

2.4. Authentification de contenu et cryptographie à clé publique

L'utilisation de la cryptographie à clé publique permet non seulement d'authentifier le contenu d'un message mais également d'authentifier l'identité de son auteur. La signature numérique d'Alice apposée sur un message M, n'est valable que pour ce message et ne peut être générée que par une personne disposant de la clé privée d'Alice . Si M est modifié, de quelque manière que ce soit, la signature ne correspondra plus.

Ainsi, la validité de la signature numérique permet de prouver deux propriétés immédiates du message M reçu :

- le message n'a pas été modifié (**authenticité du contenu**),
- l'auteur du message ne peut être qu'Alice (**authenticité d'origine**).

Contrairement au checksum qui peut être généré par n'importe qui, une signature ne peut être créée que par une personne possédant la clé privée. La vérification de la signature numérique ne demande que la possession de la clé publique. Ainsi, Alice peut signer ses messages et tout le monde peut vérifier qu'il s'agit bien de la signature d'Alice sans pour autant savoir imiter cette signature. L'emploi spécifique des méthodes de signature numérique sera traité dans le protocole suivant.

2.5. Problèmes liés à l'authentification

En général, les protocoles permettent d'identifier correctement les acteurs et de garantir l'intégrité d'un message. Cependant, aucune protection n'a été prévue jusqu'à présent pour éviter l'envoi répété du même message (le **rejeu**).

On pourrait aisément imaginer qu'Alice transmette plusieurs fois le même message M à Bob. Si M est accompagné d'un checksum et qu'il n'a pas été modifié pendant son transfert, Bob ne pourra se rendre compte qu'il a déjà reçu M. D'autant plus si Bob est un ordinateur ! Ce type de problème peut être évité en ajoutant au message, un identificateur unique de transaction (un nounce) qui rend le message unique.

Imaginons encore qu'Alice prétende qu'elle n'a jamais envoyé un message à Bob. Il s'agit de la **répudiation** d'origine.

Ce sont les protocoles de signature numérique qui permettront d'éviter les problèmes de rejeu et d'assurer la non-répudiation des documents.

3. *Protocole de signature numérique*

L'utilisation de la signature numérique apporte un plus dans les protocoles d'authentification puisqu'elle garantit l'authenticité du contenu et de l'origine d'un message.

Lors de l'examen des cryptosystèmes asymétriques, nous avons vu qu'ils possédaient les propriétés suivantes :

$M'' = D_{skA}(M) = S_{skA}(M)$ pour signature (authentification d'origine)
$M = E_{pkA}(M'') = V_{pkA}(M'')$ pour vérification (authentification de contenu)
$D_{skA}(E_{pkA}(M)) = M = V_{pkA}(S_{skA}(M)) = E_{pkA}(D_{skA}(M))$

Si l'on a appliqué un déchiffrement avec la clé privée sk_A sur le message M en clair pour obtenir M'', on peut retrouver le message M en clair en effectuant une opération de chiffrement à l'aide de la clé publique pk_A. Celui qui connaît la clé publique peut donc récupérer le texte en clair, alors que seul le détenteur de la clé privée peut avoir généré le cryptogramme.

Nous parlons de signature puisqu'une seule personne peut effectuer la transformation vers le cryptogramme et que tout le monde est en mesure de valider cette signature (figure 3.6.). Une signature numérique sera considérée comme valide si le texte en clair calculé par l'opération de vérification est un texte valide (s'il a un sens pour le destinataire).

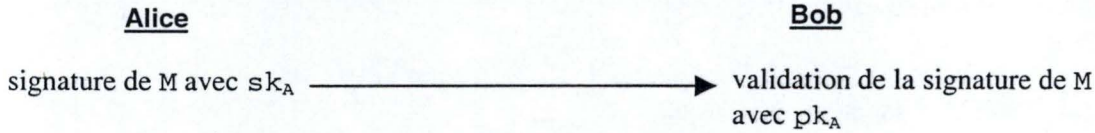

Figure 3.6. : signature asymétrique simple.

Nous utiliserons, dans la partie pratique de ce travail, une variante à ce protocole (figure 3.7.). Il ne s'agit plus de signer le texte dans son entièreté mais de signer le résumé qui lui correspond. Dans ce cas, Alice doit envoyer simultanément le message M (en clair) et la signature de résumé h(M) correspondant. Bob, le destinataire, devra effectuer deux opérations à la réception du couple (M, signature).

D'abord, il lui faudra valider la signature du résumé (c.-à-d. chiffrer la signature avec la clé publique pk_A d'Alice). Ensuite, Bob devra calculer le résumé h(M') du message M' reçu et comparer celui-ci avec le résumé h(M) reçu.

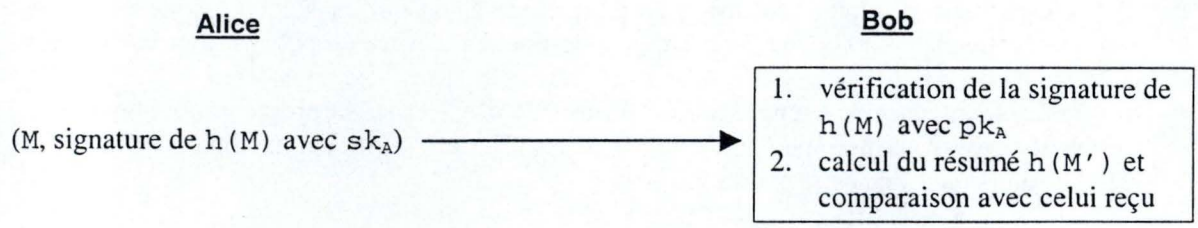

Figure 3.7. : signature numérique et fonctions à sens unique.

4. *Protocole de gestion des clés.*

Tous les protocoles que nous avons présentés jusqu'ici sont basés sur la détention d'un secret, les clés. Il est donc primordial d'assurer :

- la confidentialité et l'intégrité des **clés privées** ou **secrètes**,
- la distribution des **clés publiques**.

4.1. Stockage des clés privées et des clés secrètes

Quels que soient les algorithmes cryptographiques (symétrique ou asymétrique) que nous utilisons, il faut se poser la question du stockage des clés. **Clés privées** pour les cryptosystèmes asymétriques et **clés secrètes** pour les cryptosystèmes symétriques doivent être conservées de telle manière que leur confidentialité et leur intégrité soient garanties.

Notons que le problème du stockage des clés est un problème à part entière et qu'il dépend de la sécurité des systèmes d'exploitation.

Dans ce travail, nous n'avons pas eu l'occasion de nous pencher sur ce problème mais nous avons toutefois envisagé deux solutions de sorte que la sécurité des clés ne soit pas entièrement entre les mains du système d'exploitation. :

- stockage dans la base des registres de l' OS (`registry`) ;
- stockage sur carte magnétique.

Chacune de ces deux solutions dépend de l'implémentation pratique des outils de sécurité utilisés par le système. Cette implémentation sera traitée dans la seconde partie de ce mémoire.

4.2. Autorité de certification

Pour assurer l'efficacité du chiffrement asymétrique, la **clé publique** doit être connue de tous par l'affichage de cette clé dans un répertoire électronique. De plus, il importe d'avoir la certitude que la clé publique n'a pas été trafiquée, parce que :

- Si la clé publique d'Alice a été modifiée de quelque façon que ce soit, personne ne pourra employer cette clé pour chiffrer un message à destination d'Alice ou pour vérifier la signature d'Alice.
- Si Charles – un fraudeur - remplace la clé publique d'Alice par sa propre clé publique et si Bob veut envoyer un message confidentiel à Alice, Charles pourra, par exemple, déchiffrer ce message alors qu'Alice ne le pourra pas.

Un **tiers de confiance** peut assurer les services de gestion des clés afin de donner la certitude que les clés publique et privée supposées être associées à Alice soient vraiment celles d'Alice. L'**autorité de certification** (CA), qui joue le rôle du tiers de confiance, prend en charge la génération et la distribution des paires de clés. Elle distribue également des certificats numériques identifiant chacun des acteurs à qui elle a associé une paire de clés.

En effet, une personne désireuse, entre autre, de signer des documents doit disposer d'une paire de clés. Il lui faudra alors contacter une autorité de certification et s'identifier auprès de celle-ci. Les informations concernant son identité seront placées dans un certificat. A ce certificat (donc à la personne qu'il identifie), l'autorité de certification associe une paire de clés. La figure 3.9. schématise un certificat et montre les informations principales stockées dans celui-ci.

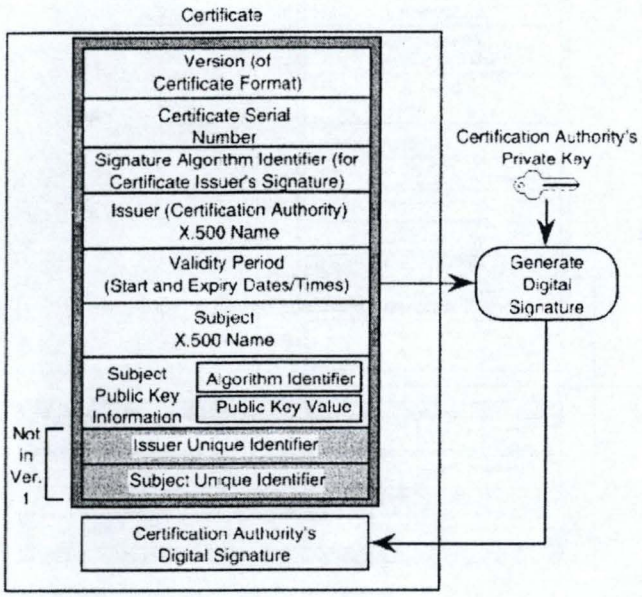

Figure 3.9. : format des certificats numériques (version 1 et 2).

Les formats des certificats sont définis par les **PKCS** (Public Key Cryptography Standards). Actuellement, le format le plus reconnu pour les certificats à clé publique est le ISO/IEC/ITU X.509. La spécification de **X.509** a été publiée dans PKCS#7.

Le format de certificat X.509 présente **trois versions** : version 1, version 2 et version 3 qui n'est qu'une extension des versions 1 et 2.

Intéressons-nous quelque peu aux informations stockées dans un certificat :

- *Version :* elle indique la version du certificat (1, 2 ou 3).
- *Serial Number :* identifiant unique assigné par la CA qui a distribué le certificat.
- *Signature :* algorithme de la signature utilisée par la CA pour signer le certificat.
- *Issuer :* nom de la CA qui a distribué le certificat.
- *Validity period :* date et heure de début et de fin de validité du certificat.
- *Subject :* nom du propriétaire dont la clé publique correspondante est certifiée.
- *Subject public key information :* valeur de la clé publique du propriétaire et l'identifiant de l'algorithme avec lequel la clé doit être utilisée.
- *Issuer unique identifier :* chaîne de caractères optionnelle utilisée pour rendre le nom de la CA non-ambigu dans le cas où le même nom aurait été réassigné à différentes entités.
- *Subject unique identifier :* chaîne de caractères optionnelle utilisée pour rendre le nom du sujet non-ambigu dans le cas où le même nom aurait été réassigné à différentes entités.

Le format d'un certificat X.509 **version 3** est présenté à la figure 3.10. A l'exception d'un champ supplémentaire pour l'extension, la version 3 a le même format que la version 2. Cette extension existe pour faciliter l'accroissement lorsqu'il y a des changements au standard.

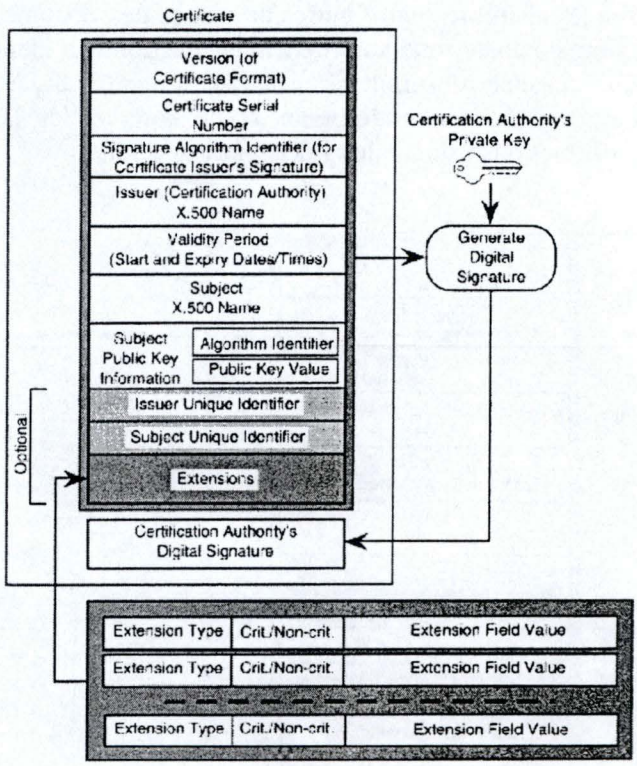

Figure 3.10. : format des certificats de version 3.

Les différentes versions de certificats se retrouvent dans les classes de certificats que proposent les autorités de certification. Par exemple, la CA `GlobalSign` offre trois types de certificats :

- Le certificat de **classe 1** est purement démonstratif et n'offre aucune garantie quant à l'identité, mais identifie juste une adresse e-mail.
- Le certificat de **classe 2** fournit une assurance quant à l'identité en exigeant une confirmation d'une tierce partie sur le nom, l'adresse et d'autres informations personnelles.
- Le certificat de **classe 3** fournit un niveau de certification encore plus haut sur l'identité, en exigeant que l'utilisateur se présente physiquement devant une autorité d'enregistrement locale (*Local Registration Authority*)

La CA a la responsabilité de mettre à jour les listes de correspondance entre certificat et clé publique. C'est auprès d'elle qu'il faut s'adresser si l'on désire demander un nouveau certificat (requête), si l'on veut révoquer un certificat ou si l'on souhaite vérifier la validité des certificats installés sur les machines. Lorsqu'un certificat n'est plus valide, il doit être supprimé.

PARTIE II : Aspects pratiques

Ce mémoire pourrait être considéré comme un projet de développement informatique à part entière. En effet, trois phases peuvent être mises en évidence : la **préparation** du projet (chapitres IV et V), le **développement** d'outils de signature (chapitre VI) et le **déploiement** des solutions dans des applications concrètes (chapitre VII). La phase de **maintenance**, quant à elle, est laissée aux bons soins de l'équipe pour laquelle ce mémoire a été réalisé.

Le présent document se veut être le guide de l'utilisateur des outils développés mais également la documentation de référence pour les améliorations à venir.

Chapitre IV : Analyse préliminaire

Dans tout projet informatique, la phase de préparation est une étape importante. Son rôle est de déterminer les objectifs de notre projet et l'utilité de celui-ci pour l'équipe PBFlow qui est à la source de ce mémoire.

Cette analyse préliminaire est divisée en quatre grandes sections : la première présentera brièvement le projet PBFlow, ses objectifs et son environnement. Les deuxième et troisième sections seront, pour leur part, consacrées respectivement au cahier de charges et à un rapide bilan des produits de signature numérique disponibles sur le marché. Finalement, nous présenterons, dans la dernière section, la classe COutils et nous justifierons nos choix de développement.

1. *Projet PBFlow*

L'équipe PBFlow (Permis de Bâtir - Flow) de l'Institut d'Informatique des Facultés Notre-Dame de la Paix à Namur a développé, à la demande de la Région Wallonne, un système d'information destiné à gérer électroniquement les demandes de permis d'urbanisme adressées à la ville de Namur.

1.1. Objectifs poursuivis

L'application PBFlow prend en charge les flux de données entre les différents acteurs participant au cycle de vie d'un permis d'urbanisme. Cette application permet d'intégrer les systèmes d'information existants dans les différentes administrations, grâce au concept de « Workgroup Data Flow Broker » et au logiciel WDFB qui l'implémente.

Le logiciel WDFB est un *système de collaboration fédérateur*, c'est-à-dire un système informatique qui simplifie les échanges d'informations entre des personnes amenées à travailler sur un même dossier, que ces personnes appartiennent à une même organisation ou non. Il crée donc une passerelle entre des systèmes informatiques a priori incompatibles. WDFB est essentiellement basé sur la notion de *dossier électronique*, et les technologies mises en jeu sont celles de l'*Internet* de Microsoft (MS-Windows NT, MS-Visual Basic, MS-SQL, MS Internet Explorer 4, ...).

Les intervenants sur un dossier de permis d'urbanisme sont nombreux et leurs relations complexes. Le projet PBFlow vise à organiser le travail et les interactions entre tous ces différents acteurs.

1.2. Environnement

A l'origine du dossier, il y a le citoyen représenté par un architecte. On trouve ensuite la ville de Namur et la DGATLP (Direction Générale de l'Aménagement de Territoire, du Logement et du

Patrimoine de la Région Wallonne) qui analysent le dossier et enfin, le Gouvernement Wallon qui joue le rôle d'arbitre lorsqu'il y a désaccord au sujet du dossier.

Le processus démarre sans aucune autre contrainte que la volonté du **demandeur**, qui prend contact avec un architecte. L'architecte introduit une demande de permis d'urbanisme auprès de la ville. Il s'agit de la naissance du dossier.

La **ville**, par l'intermédiaire du Collège des Bourgmestre et Echevins (CBE) analyse le dossier, remet un avis et envoie celui-ci à la DGATLP.

Lorsque le Fonctionnaire Délégué (FD) de la **DGATLP** reçoit un avis de la ville, il en prend connaissance et vérifie sa conformité par rapport à son propre avis.

Si la décision est conforme, il classe le dossier, sinon il prépare un recours auprès du **Gouvernement Wallon**. Si le Gouvernement Wallon aboutit à une décision dans les délais (75 jours), il envoie cette décision au demandeur, à la ville et au Fonctionnaire Délégué, sinon la décision qui a motivé le recours est confirmée (le recours est donc annulé). Dans tous les cas, le dossier est clôturé.

La figure 4.1. présente les acteurs de PBFlow et leurs interactions.

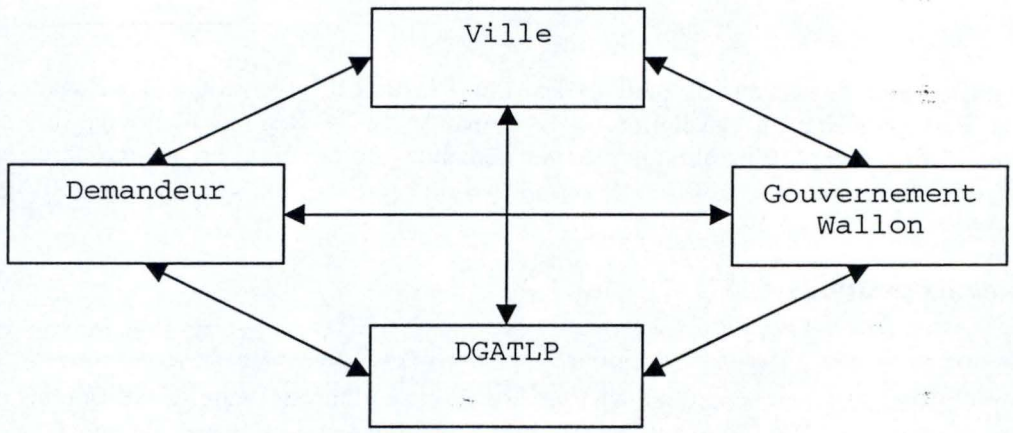

Figure 4.1. : acteurs PBFlow.

Cette trop succincte description du cycle de vie d'un permis d'urbanisme nous a permis de mettre en évidence quatre entités principales : le demandeur, la ville, la DGATLP et le Gouvernement Wallon. Ces entités, à part le demandeur, ont une organisation interne et font intervenir plusieurs de leurs services (les services administratifs et techniques de la ville par exemple ou le service administratif et le service juridique de la DGATLP). Chacun des services prenant part au cycle de vie d'un permis d'urbanisme est représenté par un acteur privilégié qui signe les avis que son service émet.

En pratique, l'utilisation de la signature manuscrite garantit l'authenticité du contenu et de l'origine de chacun des avis. Dans le cadre de l'application PBFlow, la signature manuscrite n'a pas de sens et la signature numérique n'a pas été implémentée.

C'est le but de notre projet : mettre en place des techniques informatiques pour permettre aux différents acteurs de l'application PBFlow de signer numériquement les documents qu'ils s'échangent.

2. *Cahier de charges*

Le but de notre projet est clair : il faut acquérir des outils de signature numérique permettant la signature et la vérification de signature pour tous les types de documents gérés par l'application `PBFlow`.

Dans cette section, nous présentons nos objectifs tels qu'ils ont été définis par l'équipe `PBFlow` dans le cahier de charge. Les objectifs à atteindre sont de trois types :

- signature numérique,
- vérification d'une signature,
- gestion des clés.

2.1. Signature numérique

Les outils de signature doivent permettre la signature numérique sur tous les types de documents échangés par l'application `PBFlow` (extensions `doc`, `gif`, `pdf`, `zip`, ...).

Les utilisateurs devront pouvoir choisir le certificat du signataire au cas où plusieurs certificats seraient installés sur leur machine. L'algorithme de signature devra également être sélectionné[3].

A un document en clair, il nous est demandé d'associer un fichier de signature (figure 4.2).
Ce fichier de signature contiendra les informations suivantes :

- *Signature* : il s'agit de la signature du résumé par la clé privée du signataire.
- *Certificat du signataire* : le certificat contient toutes les informations utiles au sujet du signataire, son identité, le poste qu'il occupe au sein de son organisation et sa clé publique. La raison en est qu'il nous faut la clé publique du signataire pour vérifier la signature et que le certificat n'est pas disponible éternellement auprès de la CA[4].
- *Algorithmes* : afin de garantir que toutes les informations concernant les méthodes qui ont permis de signer le document soient connues lors de la vérification, il faut ajouter l'algorithme de signature et l'algorithme de hash choisis par le signataire.

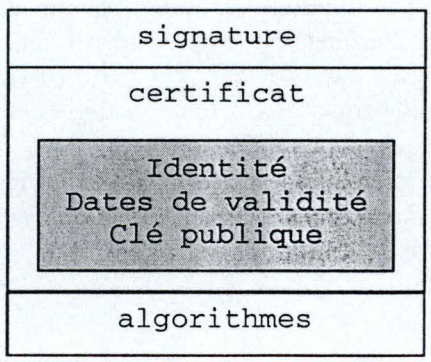

Figure 4.2. : contenu du fichier de signature.

[3] Il existe plusieurs algorithmes de signature numérique. Les utilisateurs doivent être capables de choisir le leur.
[4] Si un certificat expire, il n'est pas conservé longtemps par la CA.

2.2. Vérification d'une signature

Il est important de pouvoir vérifier la signature apposée sur un document. Cette vérification doit être une opération simple : les seuls paramètres à fournir seront le message en clair et le fichier de signature.

En cas de signature valide, il faudra afficher l'identité du signataire et les dates de validité de son certificat. Dans le cas contraire, l'utilisateur doit en être averti.

2.3. Gestion des clés

Puisque les protocoles de signature numérique et de vérification sont basés sur des clés privées et publiques, nous devrons veiller très attentivement à leur stockage et à leur emploi, plus spécialement encore pour les clés privées.

A priori, deux options sont possibles : les clés privées peuvent être stockées dans la base des registres du système d'exploitation (`registry`) ou dans des cartes magnétiques (`Smart Cards`). Il faudra envisager au minimum ces deux options.

Nous devrons porter également une attention toute particulière à la gestion des certificats. En effet, ces certificats garantissent l'emploi correct des clés. Les outils de signature devront dès lors, prendre en charge la demande, la suppression et la visualisation des certificats.

2.4. Modes d'utilisation

Une contrainte supplémentaire à respecter vient du fait que les outils de signature devront être utilisés d'une part, par un logiciel indépendant de l'application `PBFlow` et d'autre part, par l'application elle-même.

En effet, lorsque les utilisateurs créent des documents, ils devront avoir le choix de les signer soit à l'aide du logiciel indépendant, soit via l'application `PBFlow` :

- **Logiciel de signature :** l'utilisateur applique sa signature à l'aide d'un logiciel de signature installé sur sa machine. Ce logiciel lui permet, non seulement de signer et de vérifier une signature, mais aussi de gérer ses certificats. Le logiciel est indépendant de l'application `PBFlow`, il permet aux acteurs de signer des documents et de les utiliser dans un cadre autre que celui de l'application `PBFlow`.
- **Application `PBFlow` :** lorsqu'un utilisateur doit signer les documents qu'il dépose dans un dossier électronique de permis d'urbanisme, il utilise la fonction de signature proposée par l'application `PBFlow`. Dans ce cas, c'est l'application qui prend en charge la génération de la signature.

Il nous faudra donc prévoir la possibilité d'un appel des outils de signature par `PBFlow`. C'est un des points essentiels qui justifiera nos choix de développement.

3. *Evaluation des produits disponibles sur le marché*

Dans notre recherche d'outils de signature existants sur le marché de l'*Internet*, nous nous sommes intéressés aux logiciels de signature numérique et aux outils de programmation proposant des fonctions cryptographiques (`dlls`, `activeX`[1], langages de programmation, ...).

Le but n'est pas de présenter une liste des produits que nous avons trouvés et les `scores` qu'ils auraient pu obtenir[2]. Nous préférons faire le point et tirer des conclusions d'ordre général. Ces conclusions porteront respectivement sur les logiciels de signature et sur les outils de programmation.

3.1. Logiciels de signature numérique

Les logiciels sur lesquels sont basées nos conclusions, ont été trouvés sur l'*Internet*. Certains sont distribués gratuitement, d'autres sont payants et notre évaluation porte, dans ce cas, sur les versions de démonstration que nous avons pu nous procurer.

Remarquons, d'entrée de jeu, que la plupart de ces logiciels de signature sont en anglais. Ceci pose le problème de leur utilisation en Belgique où l'anglais n'est pas la langue officielle, en particulier dans les administrations communales.

Chacun des logiciels évalués a été comparé aux contraintes du cahier de charges. Les conclusions que nous avons tirées de cette évaluation sont les suivantes:

- **fonction de signature numérique :**
  Puisqu'ils se définissent tous comme la solution ultime de signature numérique, cette fonctionnalité est proposée par tous les logiciels testés. Toutefois, ils n'indiquent en général, ni les algorithmes de signature utilisés ni la technologie employée (longueur de clés, stockage des clés, ...).
- **fonction de vérification :**
  La vérification, comme la signature, est proposée par tous les logiciels évalués. L'utilisateur est toujours averti des résultats de cette vérification.
- **fonction de gestion des certificats :**
  tous les logiciels testés gèrent les certificats installés et tous donnent la possibilité d'en installer de nouveaux. Cependant, tous ces certificats sont souvent stockés dans un répertoire quelconque, de la même manière que tous les autres fichiers. De plus, il n'est jamais permis d'utiliser un certificat sur carte magnétique ou d'effacer un certificat installé sur une machine.

Globalement, les logiciels testés sont suffisants dans des cas d'utilisations non professionnelles ou à risques faibles.

La contrainte qui impose aux outils de signature d'être utilisés à la fois dans un logiciel indépendant et par l'application `PBFlow`, élimine irrémédiablement tous les logiciels que nous avons analysés. En effet, aucun de ceux-ci n'est reprogrammable et leur code n'est jamais accessible.

---

[1] les `dlls` (`dynamic link library`) et les `ActiveX` sont des modules ajoutés à une application ou à une page `WEB` afin de réutiliser des fonctionnalités programmées par quelqu'un d'autre.

[2] la liste des produits évalués est citée dans la bibliographie.

3.2. Outils de programmation

Afin de diminuer le risque dû à l'apprentissage d'un nouveau langage, nos recherches se sont portées vers les langages dont nous avions la maîtrise : Delphi, Visual Basic, C/C++ ou Java. Toutefois, seuls les langages Java et C++ offrent leurs propres fonctions de cryptographie.

Le langage Java et son API[1] de sécurité (Java security API du Symantec JDK 1.1) sont plus conviviaux mais ils ne s'interfacent pas facilement avec Visual Basic et l'application PBFlow. Le langage C++ et sa CryptoAPI offrent, quant à eux, des facilités pour développer des objets standards (objets COM[2]) accessibles au travers de dlls et d'ActiveX par la plupart des langages de programmation (Visual Basic et l'application PBFlow en particulier). Les outils de programmation les mieux adaptés à notre situation semblent donc être les outils proposés par le langage C++ de Microsoft.

Notons que nous avons également à notre disposition des exemples de codes utilisant ces outils cryptographiques et que le concepteur[3] de ces exemples accepte de nous les commenter. Notons encore que la documentation de l'API de cryptographie, la cryptoAPI, est complète et bien illustrée.

Les outils de cryptographie (la cryptoAPI de Microsoft) que nous allons utiliser seront présentés au chapitre suivant.

4. *Choix de développement*

Sur base de notre cahier de charges et de nos recherches de produits de signature existant sur l'*Internet*, nous pouvons faire les choix de développement suivants :

- Puisqu'aucun logiciel existant ne respecte entièrement notre cahier de charges, nous développerons nos propres outils de signature numérique.
- Ces outils de signature seront développés dans le langage C++ et utiliseront l'API de cryptographie proposée par Microsoft.

Les outils que nous allons développer devront, bien sûr, être utilisés à la fois par un logiciel indépendant de l'application PBFlow et par l'application elle-même. C'est pourquoi nous avons choisi de créer en C++ une classe d'objets de signature et de placer celle-ci dans une dll de sécurité.

Le choix d'une classe d'objet se justifie par le fait que nous pourrons rassembler de la sorte toutes les fonctionnalités du cahier de charges autour d'un seul et unique objet : le certificat du signataire. En effet, il est apparu que la signature et la vérification nécessitaient des informations contenues dans le certificat du signataire : le nom, le poste occupé et les dates de validité du certificat pour la signature ; le nom, le poste occupé, les dates de validité du certificat et la clé publique pour la vérification. De plus, les opérations de gestion des certificats s'appliquent également toujours sur un certificat.

---

[1] API : Application Programming Interface.

[2] COM : les Component Object Model sont des standards proposés par Microsoft qui définissent une structure et une interface pour des objets réutilisables dans plusieurs langages tels que C++, Delphi ou Visual Basic.

[3] Encore merci à M. Bruno Loodts.

Le choix de la dll, quant à lui, tient dans le fait qu'elle nous permettra d'accéder à notre classe d'objets développée en C++ à partir de l'application PBFlow développée en Visual Basic (VB) et d'un futur logiciel de signature qui sera également développé en VB.

La figure 4.3. présente nos choix de développement et l'emploi qui en sera fait : l'application PBFlow et le logiciel de signature accèdent à notre dll de sécurité implémentant la classe d'outils de signature basée sur l'emploi de la cryptoAPI de Microsoft.

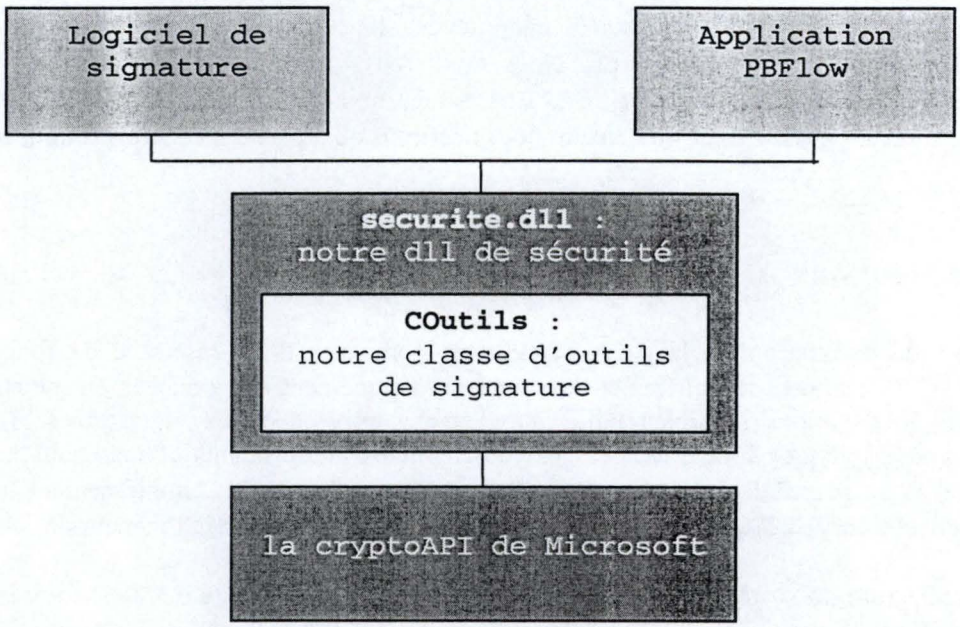

Figure 4.3. : choix de développement

Dans le chapitre suivant (chapitre V), nous mettrons un terme à la phase de préparation de notre projet en présentant la cryptoAPI. Nous montrerons comment elle est construite et comment nous pouvons l'utiliser pour réaliser des opérations de cryptographie telles que la sélection d'un certificat ou la génération d'une signature.

Il s'agira ensuite de décrire l'implémentation concrète de la classe COutils, la classe d'outils de signature que nous avons développée (chapitre VI), et de montrer comment elle est utilisée par un logiciel de signature et par l'application PBFlow (chapitre VII).

Chapitre V : cryptoAPI de Microsoft

Pour compléter notre phase de préparation, nous présentons les outils de cryptographie qui seront utilisés par la classe COutils. Ce chapitre est donc consacré à la librairie d'outils de cryptographie proposée par Microsoft. Nous étudierons, dans un premier temps, le modèle théorique sur lequel se base la cryptoAPI. Ensuite, nous définirons les Cryptographic Service Providers (CSPs) et finalement, nous décrirons quelques cas concrets d'utilisation.

1. *Modèle théorique*

L'API de cryptographie, la cryptoAPI, est composée d'un ensemble de fonctions qui permettent aux applications de chiffrer et de signer numériquement des données en garantissant la protection des informations sensibles telles que les clés privées et les algorithmes. Toutes les opérations cryptographiques sont effectuées par des modules indépendants connus sous le nom de cryptographic service providers (CSPs). Tous les CSPs implémentent de façon différente la couche cryptoAPI (au niveau des algorithmes, au niveau de la longueur des clés, ...).

L'architecture du système cryptographique de Microsoft (figure 5.1.) est divisée en trois domaines : les applications, le système d'exploitation et les CSPs. Les applications ne communiquent avec l'OS[1] qu'au travers d'un ensemble de fonctions programmées au sein de la cryptoAPI : elles ne communiquent pas directement avec les CSPs. Chacun des appels à une fonction cryptographique est dévié vers l'OS. Un paramètre dans chaque fonction de la cryptoAPI indique au système d'exploitation quel CSP utiliser et quelle opération cryptographique effectuer.

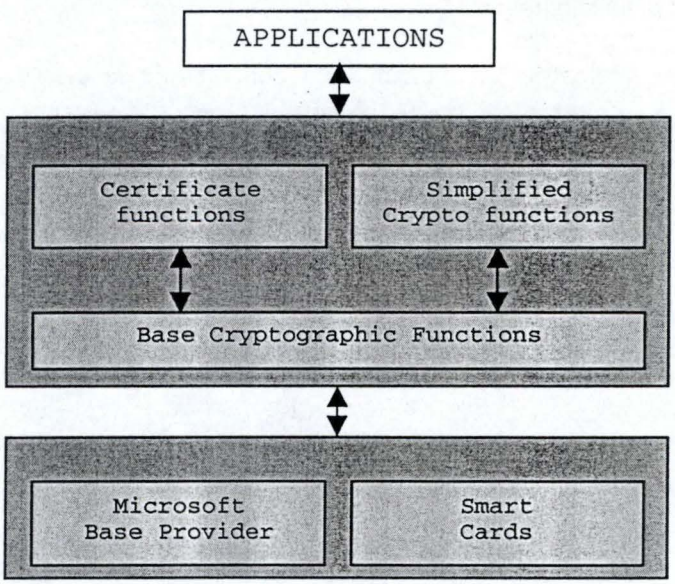

Figure 5.1. : architecture de la cryptoAPI.

[1] OS : Operating System.

Ainsi, la cryptoAPI fait écran entre l'application et les algorithmes qui protègent les données. La figure 5.1. présente l'architecture du système cryptographique. Une application fait des appels à des fonctions de cryptographie ou à des fonctions de stockage des certificats. Ces fonctions sont réalisées par des services de base accédant aux CSPs. Les fonctions cryptographiques prennent en charge le chiffrement et la signature tandis que les fonctions de stockage des certificats extraient, stockent, vérifient ou énumèrent les certificats installés sur la machine.

2. *Cryptographic Service Providers*

Les Cryptographic Service Providers sont les modules indépendants qui réalisent le vrai travail de cryptographie. Un CSP est composé d'une dll et d'un fichier de signature que la cryptoAPI utilise régulièrement pour vérifier l'intégrité et l'identité du Provider lui-même. Un CSP est un fournisseur de services cryptographiques réalisant un certain nombre de tâches bien précises invoquées par l'application via la cryptoAPI. Un CSP n'est rien d'autre qu'un module « plug and play » : une fois enregistré, il peut être utilisé sans rien modifier au niveau application.

Le CSP par défaut est le Microsoft RSA Base Provider implémenté dans la librairie rsabase.dll. Il s'agit d'un CSP du type PROV_RSA_FULL supportant l'algorithme RSA avec des clés de 512 bits. Il utilise également les algorithmes RC2 et RC4 sur des clés de 40 bits pour le chiffrement symétrique et le SHA pour le calcul des résumés.

Chaque CSP est associé à une base de données de containers de clés dans lesquels sont stockées toutes les clés privées et publiques de tous les utilisateurs accédant à la machine (figure 5.2.). Chaque container est identifié par un nom unique. Par défaut, cet identifiant correspond au nom de logon[1] de chaque utilisateur.

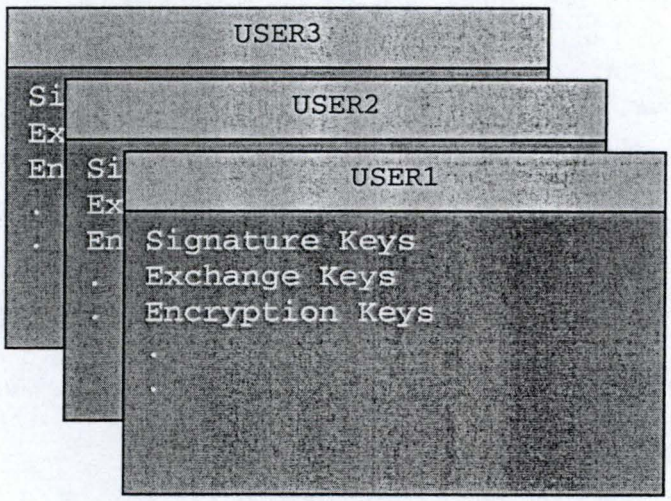

Figure 5.2. : base de données des containers de clés.

[1] le nom de logon est le nom sous lequel l'utilisateur s'identifie lorsqu'il ouvre une session Windows NT.

Pour assurer leur caractère secret, toutes les données manipulées par un CSP (spécialement les clés) sont retournées à l'appelant sous la forme d'un handle[1] et restent inaccessibles au niveau de l'application. Les programmes se limitent à passer des données et à spécifier le type de fonction cryptographique à effectuer.

3. *Cas concrets d'utilisation*

Dans cette section, nous présentons trois cas concrets d'utilisation de la cryptoAPI : la connexion à un CSP, la sélection d'un certificat et la signature numérique. Notre but n'est pas de détailler chacun des appels aux fonctions de la cryptoAPI mais bien de mettre en évidence la manière dont ces fonctions peuvent être utilisées pour réaliser des tâches concrètes de cryptographie.

Notons que les codes présentés ne sont pas complets et que les fonctions ou les paramètres importants sont indiqués en caractères gras.

Notons encore que les fonctions de cryptographie et les fonctions de stockage de certificats proposées par la cryptoAPI sont implémentées respectivement dans advapi32.dll et dans crypt32.dll et qu'elles sont accessibles depuis C++ via le fichier de ressources wincrypt.h.

3.1. Initialisation et connexion à un CSP

Avant tout appel à une fonction de la cryptoAPI, il faut choisir le module qui réalisera le véritable travail de cryptographie (un CSP particulier). La fonction CryptAcquireContext est utilisée pour obtenir un handle vers un container de clés particulier au sein d'un CSP particulier. Cet handle sera utilisé à chaque appel de fonction. La fonction CryptReleaseContext permet de libérer le handle retourné par la fonction CryptAcquireContext.

CryptAcquireContext effectue deux opérations. Premièrement, elle tente de trouver un CSP correspondant aux paramètres qui lui sont passés. Si un CSP est trouvé, la fonction recherche, dans ce Provider, le container de clé demandé.

Pour obtenir un handle vers le CSP par défaut, nous écrirons :

```
#define MS_DEF_PROV         « Microsoft Base Cryptographic Provider v1.0 »
#define PROV_RSA_FULL       1

LONG          lgResult ;
HCRYPTPROV    hProv ;       //handle to CSP

//attempt to acquire a handle to the default CSP
lgResult = CryptAcquireContext(
      &hProv,               //handle to CSP
      NULL,                 //use default key container
      MS_DEF_PROV,          //use default CSP
      PROV_RSA_FULL,        //type of provider to acquire
      0);                   //not used here
```

[1] un handle est un pointeur particulier à partir duquel l'application référence une zone mémoire mais ne peut obtenir le contenu de la zone mémoire pointée.

Si l'appel à la fonction CryptAcquireContext est un succès, la variable hProv contiendrait le handle vers le container de clés par défaut, dans le CSP par défaut.

Lorsque l'on désire libérer un handle précédemment acquis, nous écrirons :

```
//release handle to CSP
CryptReleaseContext(hProv,0);
```

3.2. Sélection d'un certificat

Les certificats sont stockés par la cryptoAPI dans les répertoires de la base de registres du système d'exploitation (la registry). A chacun de ces certificats est associée une paire de clés (clé privée et clé publique) qui sera conservée à l'intérieur d'un container de clés au sein d'un CSP.

A l'instar des paires de clés qui sont conservées dans des containers de clés (key containers), les certificats sont stockés dans des containers de certificats appelés les system certificate stores.

Les system certificate stores sont de trois types :

- CA pour les certificats des autorités de certification,
- MY pour les certificats de l'utilisateur courant,
- SPC pour les certificats des éditeurs de programmes.

Ainsi, en explorant les branches de la base de registres, nous avons accès au contenu binaire des certificats installés. Les certificats de l'utilisateur courant sont stockés dans :

HKEY_CURRENT_USER\Software\Microsoft\SystemCertificates\My

La figure 5.3. montre le contenu d'un certificat. Nous remarquerons que ce certificat est associé au CSP par défaut, le Microsoft Base Cryptographic Provide v1.0.

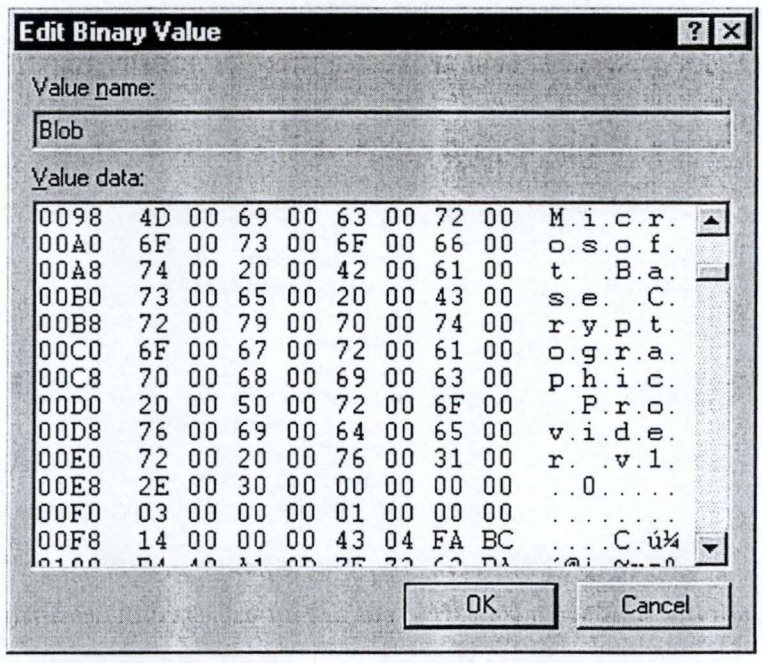

Figure 5.3. : contenu binaire d'un certificat.

La fonction permettant d'ouvrir les system stores est CertOpenSystemStore. Pour appeler cette fonction, il faut écrire :

```
HCRYPTPROV    hProv;                    //handle to CSP acquire earlier
HCERTSTORE    hCertStore = NULL;        //handle to certificate store

//open the MY system certificate store
hCertStore = CertOpenSystemStore(hProv,« MY »);
```

Si la valeur de hCertStore est différente de NULL, hCertStore est le handle vers le system store dans lequel sont stockés les certificats de l'utilisateur courant.

Dès que le store est ouvert, nous pouvons parcourir la liste des certificats qui y sont stockés. La fonction accédant à ces certificats est CertEnumCertificatesInStore. Ses appels ont la forme :

```
PCCERT_CONTEXT  pPrevCertContext = NULL;      //pointer to certificate
PCCERT_CONTEXT  pCertContext     = NULL;      //pointer to certificate
HCERTSTORE      hCertStore;                   //handle to certificate store

pCertContext = CertEnumCertificatesInStore(hCertStore,pPrevCertContext);
```

Si pCertContext diffère de NULL après l'appel, pCertContext pointe vers le premier certificat stocké dans le system store (hCertStore). Si pPrevCertContext pointait vers un certificat, pCertContext serait un pointeur vers le certificat suivant pPrevCertContext dans la liste.

Dans le fichier de ressources wincrypt.h implémentant les appels par C++, aux fonctions de la cryptoAPI, le type PCERT_CONTEXT est définit comme suit :

```
//Certificate context
typedef struct _CERT_CONTEXT {
      DWORD         dwCertEncodingType;     //encoding type
      BYTE          *pbCertEncoded;         //encoded data
      DWORD         cbCertEncoded;          //length of encoded data
      PCERT_INFO    pCertInfo;              //certificate infos
      HCERTSTORE    hCertStore;             //system store
} CERT_CONTEXT, *PCERT_CONTEXT;
typedef const CERT_CONTEXT *PCCERT_CONTEXT;
```

La variable la plus intéressante pour nous est la structure PCERT_INFO contenant toutes les informations stockées dans un certificat :

```
//Information stored in a certificate
typedef struct _CERT_INFO {
      DWORD                        dwVersion;
      CRYPT_INTEGER_BLOB           SerialNumber;
      CRYPT_ALGORITHM_IDENTIFIER   SignatureAlgorithm;
      CERT_NAME_BLOB               Issuer;
      FILETIME                     NotBefore;
      FILETIME                     NotAfter;
      CERT_NAME_BLOB               Subject;
      CERT_PUBLIC_KEY_INFO         SubjectPublicKeyInfo;
      CRYPT_BIT_BLOB               IssuerUniqueId;
      CRYPT_BIT_BLOB               SubjectUniqueId;
      DWORD                        cExtension;
      PCERT_EXTENSION              rgExtension;
} CERT_INFO, *PCERT_INFO;
```

Bien que les informations soient stockées sous des formats spécifiques à la cryptoAPI, nous pouvons aisément les mettre en relation avec les données contenues dans un certificat de classe 3.

En particulier, notons la présence des noms du propriétaire (subject) et de la CA (issuer), de la clé publique (SubjectPublicKeyInfo) et des dates de validité du certificat (NotBefore et NotAfter).

3.3. Signature numérique

La signature numérique doit correspondre, selon notre cahier de charge, à la création d'un fichier de signature associé à un message en clair et séparé de celui-ci. Cette signature sera générée à partir de la clé privée du signataire. Nous rappelons les informations indispensables devant se trouver dans le fichier de signature :

- *Signature* : la signature du résumé du message par la clé privée du signataire,
- *Certificat* : le certificat contient toutes les informations utiles au sujet du signataire,
- *Algorithmes* : l'algorithme da signature et l'algorithme de hash utilisés.

La procédure à suivre pour signer des données est assez complexe et comporte quatre étapes :

1. Choix du CSP qui générera la signature,
2. Sélection du certificat du signataire,
3. Acquisition du handle vers le container contenant la clé privée associée à ce certificat,
4. Signature des données.

Nous avons déjà traité les deux premier cas. Intéressons-nous à la sélection d'un container de clé et à la signature proprement dite.

3.3.1. Sélection d'un container de clé en fonction d'un certificat

Il ne s'agit plus de choisir le container par défaut comme nous l'avons fait précédemment. Il faut obtenir un handle vers le container contenant la clé privée associée au certificat du signataire. Dans ce cas, nous devons utiliser les informations stockées dans le certificat du signataire. Supposons que ce certificat ait déjà été sélectionné. Nous écrirons alors :

```
LONG                        lgResult;
PCCERT_CONTEXT              pSignerCert;              //chosen certificate
CRYPT_KEY_PROV_INFO*        pCryptKeyProvInfo;        //private key infos
DWORD                       cbData;                   //length of buffer
HCRYPTPROV                  hProv;                    //handle to CSP

//get the properties of the Signer Certificate
//get the size of the properties
lgResult = CertGetCertificateContextProperty(
      pSignerCert,                    //signer certificate
      CERT_KEY_PROV_INFO_PROP_ID,     //get the private key infos
      NULL,                           //not used here
      &cbData);                       //length buffer

//allocate memory for private key properties
pCryptKeyProvInfo = (CRYPT_KEY_PROV_INFO*)malloc(cbData);

//get the properties of the Signer Certificate.
lgResult = CertGetCertificateContextProperty(
      pSignerCert,                    //signer certificate
      CERT_KEY_PROV_INFO_PROP_ID,     //get the private key infos
      pCryptKeyProvInfo,              //private key infos
      &cbData);                       //length of buffer
```

```
//get the container associated with the private key using private key infos.
lgResult = CryptAcquireContext(
      &hProv,                                          //handle to CSP
      pCryptKeyProvInfo->pwszContainerName,            //key container
      pCryptKeyProvInfo->pwszProvName,                 //csp to select
      pCryptKeyProvInfo->dwProvType,                   //provider type
      pCryptKeyProvInfo->dwFlags);
```

Le premier appel à la fonction `CertGetCertificateContextProperty` nous permet de calculer la place mémoire à allouer à la structure contenant les informations sur la clé privée du signataire. La fonction `malloc` réserve de la place en mémoire et le second appel à la fonction `CertGetCertificateContextProperty` y stocke les données.

Ainsi, nous avons obtenu un `handle` vers le container de clés correspondant au certificat du signataire. Lors de la génération de la signature, c'est cet `handle` qui sera référencer pour indiquer la clé privée à utiliser.

3.3.2. Signature numérique de données

Nous considérons le cas de la signature des données contenues dans la variable `pbContent` et de longueur égale à `cbContent`.

Avant de signer les données, il faut encore paramétrer la procédure de signature en choisissant, entre autres, l'algorithme de signature et l'algorithme de hash, le certificat du signataire et le container de clés associé. Nous écrirons :

```
//encoding types for certificates
#define MY_ENCODING_TYPE  (PKCS_7_ASN_ENCODING | X509_ASN_ENCODING)

LONG                lgResult;

//create the MessageArray to be signed using pbContent & cbContent
const BYTE*         MessageArray[] = {*pbContent};   //data to be signed
DWORD               MessageSizeArray[1];             //length of data
MessageSizeArray[0] = *cbContent;

//create a MsgCertArray containing the signer certificate
PCCERT_CONTEXT      MsgCertArray[1];
MsgCertArray[0] = pSignerCert;                       //signer certificate

//initialize the Algorithm Identifier structure
CRYPT_ALGORITHM_IDENTIFIER      HashAlgorithm;       //algorithm to hash
DWORD                           HashAlgSize;         //length of algorithm
HashAlgSize = sizeof(HashAlgorithm);                 //get the size
memset(&HashAlgorithm, 0, HashAlgSize);              //allocate memory

//SHA is the chosen hash algorithme & RSA is used to generate the signature
HashAlgorithm.pszObjId = szOID_RSA_SHA1RSA;

//initialize the signature structure
CRYPT_SIGN_MESSAGE_PARA   SigParams;                 //signature parameters
DWORD                     SigParamsSize = sizeof(SigParams);
memset(&SigParams, 0, SigParamsSize);                //allocate memory

//set the structure values
SigParams.cbSize = SigParamsSize;                    //size
SigParams.dwMsgEncodingType = MY_ENCODING_TYPE;      //encoding types
SigParams.pSigningCert = pSignerCert;                //signature certificate
SigParams.HashAlgorithm = HashAlgorithm;             //hash algorithm
SigParams.cMsgCert = 1;
SigParams.rgpMsgCert = MsgCertArray;
```

```
//sign the message
BYTE**        pbEncryptedBlob;                    //signature
DWORD*        cbEncryptedBlob;                    //length of signature
*cbEncryptedBlob = NULL;

//get the size of the output SignedBlob.
lgResult = CryptSignMessage(
      &SigParams,                    //signature parameters
      true,                          //detached signature
      1,                             //number of messages
      MessageArray,                  //messages to be signed
      MessageSizeArray,              //size of messages
      NULL,                          //buffer for signed msg
      cbEncryptedBlob);              //size of buffer

//allocate memory for the signed blob.
*pbEncryptedBlob = (BYTE*)malloc(*cbEncryptedBlob);
memset(*pbEncryptedBlob, 49, *cbEncryptedBlob);

//get the signature file
lgResult = CryptSignMessage(
      &SigParams,                    //signature parameters
      true,                          //detached signature
      1,                             //number of messages
      MessageArray,                  //messages to be signed
      MessageSizeArray,              //size of messages
      *pbEncryptedBlob,              //buffer for signed msg
      cbEncryptedBlob);              //size of buffer
```

A la suite de ces appels, le paramètre `pbEncryptedBlob` contient toutes les informations devant se trouver dans notre fichier de signature : la signature, le certificat du signataire (`SigParams.pSigningCert`) et l'algorithmes de signature et de hash (`SigParams.HashAlgorithm`).

Il nous reste à écrire le contenu de `pbEncryptedBlob` dans un fichier séparé du message et notre fichier de signature est conforme aux contraintes du cahier de charges.

Chapitre VI : Implémentation de la classe COutils

D'aucun pourrait critiquer l'aspect trop pratique de ce chapitre mais nous estimons qu'il a sa raison d'être dans la mesure où ce document devrait servir de guide de référence pour la maintenance et les développements futurs de notre projet.

Le but est de montrer comment les outils de cryptographie (notre classe COutils) peuvent être utilisés par les applications et comment nous avons procédé pour implémenter ceux-ci. Ce chapitre correspond à la phase de développement.

1. *Introduction*

Afin d'offrir à notre classe d'outils la possibilité d'être utilisée par plusieurs langages de programmation, nous avons choisi de la placer dans une dll.

Une dll est un module indépendant (une librairie) offrant un certain nombre de services aux applications qui l'emploient. Ces services sont accessibles de l'extérieur au travers d'une interface correspondant, dans notre cas, aux définitions des fonctions de cryptographie implémentées dans la classe COutils. La figure 6.1. schématise cette architecture.

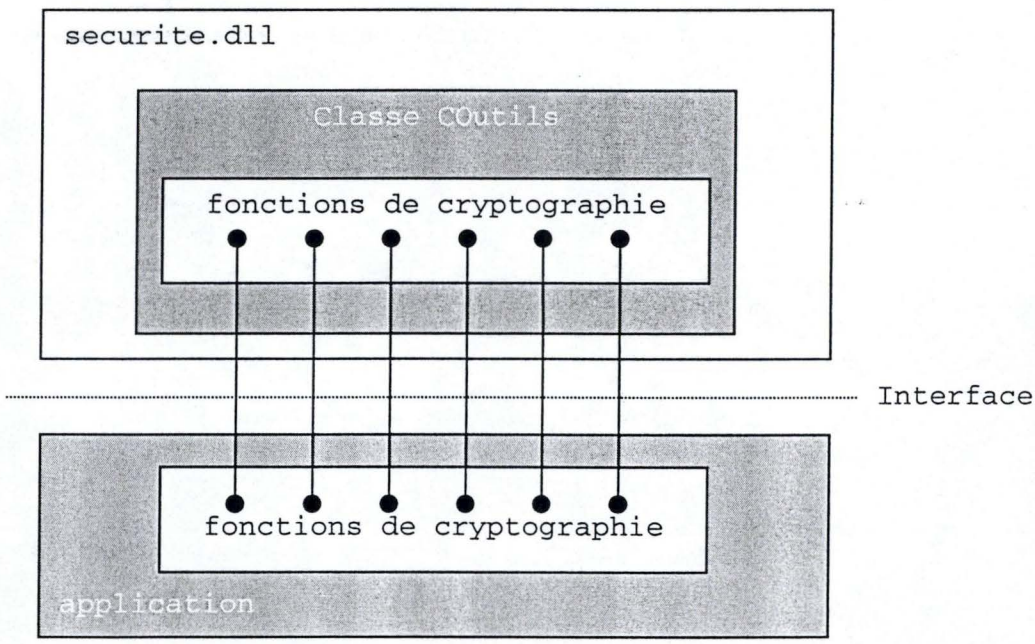

Figure 6.1. : Interface de notre dll.

Dans la suite de ce chapitre, nous proposons deux niveaux d'analyse : le premier s'attachera à définir la manière dont la classe COutils implémente l'interface de la dll et le second s'intéressera à l'implémentation des outils de cryptographie au sein de notre classe.

2. *Niveau 1 : implémentation de l'interface*

Le premier niveau d'analyse a pour but de décrire l'interface implémentée par notre classe. Cette interface correspond à l'image qu'une application se fait de la classe lorsqu'elle l'utilise.

Toutes les informations contenues dans cette section devraient être suffisantes pour garantir un usage correct de la classe COutils par une application. Il s'agit, en quelque sorte, du guide de l'utilisateur[1].

L'interface a deux rôles distincts :

1. définir précisément les services offerts par la classe aux applications,
2. permettre l'utilisation d'objets standards COM.

Une définition précise des services offerts nous permet de contrôler les emplois qui seront faits des outils de cryptographie : les utilisateurs ne pourront effectuer que les opérations autorisées et rien de plus. L'emploi des objets standards COM garantit que les outils que nous avons développés seront utilisables par un grand nombre d'applications[2].

Les services offerts par la classe COutils correspondent à la consultation des **attributs** de la classe et aux appels des différentes **méthodes** de celle-ci. La figure 6.2. représente la structure de la classe COutils telle qu'elle peut être perçue par une application au travers de l'interface.

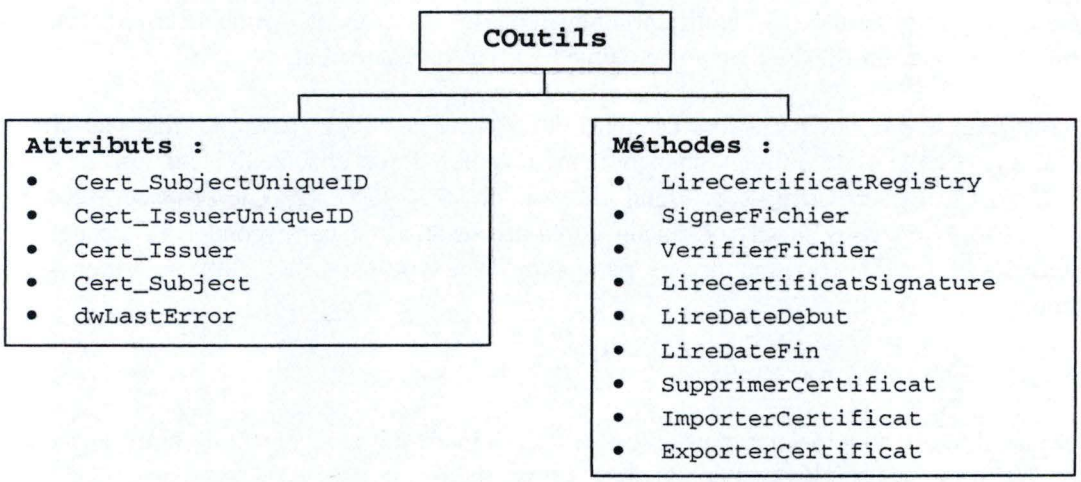

Figure 6.2. : structure de la classe COutils au travers de l'interface.

2.1. Attributs

Quatre des attributs accessibles par une application utilisant la classe COutils correspondent à une partie des informations identifiant un certificat. Ils sont de type BSTR, le type String des objets compatibles COM.

---

[1] La section suivante sera considérée plutôt comme le guide du développeur d'outils de cryptographie. La différence entre ces deux sections tient dans le niveau de description du code produit dans le cadre de ce travail.

[2] Au minimum toutes les applications développées dans un langage de programmation Microsoft sont susceptibles d'être rendues compatibles COM.

Ces attributs sont :

- BSTR Cert_Subject;
- BSTR Cert_Issuer;
- BSTR Cert_SubjectUniqueID;
- BSTR Cert_IssuerUniqueID;

Dans nos choix de développement, nous avons justifié l'utilisation d'une classe par le fait que toutes les opérations de cryptographie définies dans le cahier de charges s'appliquaient sur un certificat ou utilisaient les informations d'un certificat.

Afin de déterminer de manière sûre le certificat sur lequel ou à partir duquel vont s'appliquer les outils de notre classe (ses méthodes), nous permettons à l'application de connaître le nom[1] du propriétaire du certificat (Cert_subject) et le nom de la CA qui l'a émis (Cert_Issuer). A cela, nous ajoutons encore les identifiants uniques du propriétaire (Cert_SubjectUniqueID) et de la CA (Cert_IssuerUniqueID).

Une application voulant accéder à l'attribut Cert_SubjectUniqueID utilise, sans le savoir, la fonction de conversion de type suivante :

```
STDMETHODIMP COutils::get_Cert_IssuerUniqueID(BSTR *pVal)
{
        CComBSTR bstrRtnVal = Cert_IssuerUniqueID;
        *pVal = bstrRtnVal.Detach();
        return S_OK;
}
```

Cette fonction retourne l'identifiant unique de la CA sous la forme d'un String mais compatible COM c.-à-d. un BSTR. Il en va de même pour les autres attributs.

Nous n'en avons pas parlé lors de notre description de la cryptoAPI mais les fonctions implémentées par celle-ci retournent un paramètre d'erreur après chacun de leur appel. Ainsi, la fonction CryptAcquireContext[2] rend la valeur SUCCESS=1 en cas de réussite et NTE_BAD_KEYSET=4 dans le cas où aucun container de clés ne correspond à la clé demandée. L'attribut DwLastError correspond à ce paramètre. Son type est LONG, le type entier long des objets compatibles COM.

- LONG DwLastError;

Cet attribut est principalement utile lors des sessions de développement de nouveaux outils. Il ne sera que très rarement employé en dehors de ces sessions[3].

2.2. Méthodes

Les méthodes de la classe COutils accessibles par les applications sont de deux types : les fonctions cryptographiques et les fonctions de gestion de certificats.

Les fonctions cryptographiques permettent de signer un document et de vérifier une signature.

---

[1] Nous renvoyons aux sections 4.2. du chapitre III et 3.2. du chapitre V, le lecteur désireux de vérifier l'appartenance de ces attributs au contenu d'un certificat. Notons que le SerialNumber n'est pas utile à notre niveau.

[2] La fonction CryptAcquireContext acquiert un handle vers un CSP et vers un container de clé (voir la section 3.1. du chapitre V).

[3] En mode d'utilisation normal de la classe, cet attribut n'est d'aucune utilité.

Les fonctions de gestion de certificats donnent accès à la sélection d'un certificat installé[1], à la sélection du signataire d'un document (via le fichier de signature), à la suppression d'un certificat, à l'importation et à l'exportation d'un certificat dans un fichier et à la lecture des dates de validité d'un certificat.

Puisque ces fonctions d'interface ne servent qu'à la définition des opérations possibles et à l'utilisation de types standards, leur implémentation est très simple : chacune de ces fonctions fait appel à une des méthodes internes à la classe COutils. Si cette méthode interne n'utilise pas exclusivement des types standards, il faudra envisager des conversions. Le choix de la méthode à appeler dépend du rôle de la fonction d'interface : toutes les méthodes de la classe ne seront donc pas nécessairement accessibles par l'application.

Les arguments de ces fonctions sont tous compatibles COM. Aux types que nous avons déjà mentionnés, ajoutons le type LPSTR correspondant aux pointeurs vers des chaînes de caractères.

Pour chacune des méthodes accessibles, nous spécifierons dans le langage naturel, les pré- et les post-conditions : cela devrait suffir à garantir un usage correct des services(méthodes) proposé(e)s.

Notons que chacune des fonctions de l'interface retourne un paramètre d'erreur (lgResult). Ce paramètre indique le nom de la fonction de la cryptoAPI ayant échoué. Pour s'assurer de la réussite des appels à une de ces fonctions, l'utilisateur se contentera de tester l'égalité entre lgResult et la valeur 1 (SUCCESS). Toutes les autres valeurs retournées (lgResult) sont utilisées au cours de sessions de développement (debugging).

2.2.1. LireCertificatRegistry

Cette fonction parcourt la liste des certificats installés dans la base des registres[2]. Lorsqu'un certificat est sélectionné, il est immédiatement pointé[3] par la classe COutils et les attributs de celle-ci correspondent à ce certificat.

En-Tête :
```
LireCertificatRegistry( LONG number,          //in
                        LPSTR Store,          //in
                        LONG *lgResult);      //out
```

Pré-conditions :

- Le paramètre number est un entier supérieur ou égal à zéro. Il donne le numéro du certificat dans la liste des certificats installés.
- Le paramètre Store définit si le certificat appartient :
  - à l'utilisateur (Store = «MY»),
  - à une autorité de certification (Store = «CA»),
  - à un éditeur de programme (Store = «SPC»).

Post-conditions :

- Le paramètre lgResult vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans ce cas, les attributs de la classe sont mis à jour avec les informations du certificat choisi et COutils est liée au certificat.

---

[1] Nous verrons que les certificats seront stockés soit dans la base des registres soit sur carte magnétique.

[2] Nous verrons à la section 3.1. de ce chapitre que cette fonction tient également compte des certificats stockés sur carte magnétique dans la mesure où ces derniers sont également stockés dans la registry (aucune clé privée n'y est toutefois installée).

[3] Un attribut non accessible par l'application correspond au handle vers le certificat sélectionné (voir le paragraphe 3.4.2 du chapitre VI).

Détails d'implémentation :

```
STDMETHODIMP COutils::LireCertificatRegistry( LONG number,
                                              LPSTR StoreName,
                                              LONG *lgResult)
{
        *lgResult = GetRegistredCertificate(number,StoreName);
        return S_OK;
}
```

2.2.2. SignerFichier

Cette fonction est utilisée pour signer un document (szSourceFile) de n'importe quel type. Le signataire du document et la clé privée de signature correspondent à la personne identifiée par le certificat pointé par la classe COutils. Nous ajouterons ce certificat au fichier de signature (szSignatureFile).

En-Tête :
```
SignerFichier(LPSTR szSourceFile,                    //in
              LPSTR szSignatureFile,                 //out
              LONG *lgResult);                       //out
```

Pré-conditions :

- Le fichier à signer (szSourceFile) doit exister.
- La classe COutils doit pointer vers un certificat.

Post-conditions :

- Le fichier de signature (szSignatureFile) contient la signature du résumé de szSourceFile, le certificat pointé par la classe et l'identifiant des algorithmes utilisés (signature et hash). Si le fichier de signature existe, son contenu est remplacé. Dans le cas contraire, il est créé.
- Le paramètre lgResult vaut 1 (SUCCESS) si l'opération s'est déroulée correctement.

Détails d'implémentation :

```
STDMETHODIMP COutils::SignerFichier( LPSTR szSourceFile,
                                     LPSTR szSignatureFile,
                                     LONG *lgResult)
{
        *lgResult = SignFile(szSourceFile,szSignatureFile);
        return S_OK;
}
```

2.2.3. VerifierFichier

Cette fonction de vérification de signature permet de tester la correspondance entre une signature et un fichier en clair. Si la vérification est un succès, le fichier de signature (szSignatureFile) contient la signature du fichier szSourceFile.

En-Tête :
```
VerifierFichier( LPSTR szSourceFile,                 //in
                 LPSTR szSignatureFile,              //in
                 LONG *lgResult);                    //out
```

Pré-conditions :

- Le fichier (szSourceFile) sur lequel est supposé porté la signature, doit exister.
- Le fichier de signature (szSignatureFile) doit exister.

Post-conditions :

- Le paramètre `lgResult` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans ce cas, la signature peut être considérée comme valide.

Détails d'implémentation :

```
STDMETHODIMP COutils::VerifierFichier(LPSTR szSourceFile,
                                      LPSTR szSignatureFile,
                                      LONG *lgResult)
{
        *lgResult = VerifyFile(szSourceFile,szSignatureFile);
        return S_OK;
}
```

2.2.4. LireCertificatSignature

Cette fonction parcourt le fichier de signature (`szSignatureFile`) et en extrait le certificat du signataire. Si un certificat est trouvé, il est pointé par la classe `COutils` : les attributs de cette classe correspondent à ce certificat.

En-Tête :
```
LireCertificatSignature( LPSTR szSignatureFile,      //in
                         LONG *lgResult);            //out
```

Pré-conditions :

- Le fichier de signature (`szSignatureFile`) doit exister.

Post-conditions :

- Le certificat contenu dans le fichier de signature est pointé par la classe et les attributs de celle-ci sont mis à jour.
- Le paramètre `lgResult` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement.

Détails d'implémentation :

```
STDMETHODIMP COutils::LireCertificatSignature(LPSTR szSignatureFile,
                                              LONG *lgResult)
{
        *lgResult = GetCertificateFromSignature(szSignatureFile);
        return S_OK;
}
```

2.2.5. LireDateDebut

La fonction `LireDateDebut` permet d'accéder aux deux paramètres[1] qui déterminent la date de début de validité du certificat pointé par la classe[2].

En-Tête :
```
LireDateDebut(LONG *pValHigh,                        //out
              LONG *pValLow);                        //out
```

Pré-conditions :

- La classe `COutils` doit pointer vers un certificat.

---

[1] Les dates sont de type `FILETIME` non compatible `COM`. Il faudra donc que l'utilisateur recrée une structure `FILETIME` dans son application.

[2] L'utilité de cette fonction est indiquée à la section 2.2. du chapitre IV.

Post-conditions :

- Les paramètres pValHigh et pValLow déterminent un objet du type FILETIME correspondant à la date de début de validité du certificat pointé par COutils.

Détails d'implémentation :

```
STDMETHODIMP COutils::LireDateDebut(LONG *pValHigh ,LONG *pValLow)
{
        *pValHigh = Cert_NotBefore.dwHighDateTime;
        *pValLow = Cert_NotBefore.dwLowDateTime;
        return S_OK;
}
```

2.2.6. LireDateFin

La fonction LireDateFin permet d'accéder aux deux paramètres qui déterminent la date de fin de validité du certificat pointé par la classe.

En-Tête :
```
LireDateFin(LONG *pValHigh,                    //out
            LONG *pValLow);                    //out
```

Pré-conditions :

- La classe COutils doit pointer vers un certificat.

Post-conditions :

- Les paramètres pValHigh et pValLow déterminent un objet du type FILETIME correspondant à la date de fin de validité du certificat pointé par COutils.

Implémentaton :

```
STDMETHODIMP COutils::LireDateFin(LONG *pValHigh ,LONG *pValLow)
{
        *pValHigh = Cert_NotAfter.dwHighDateTime;
        *pValLow = Cert_NotAfter.dwLowDateTime;
        return S_OK;
}
```

2.2.7. SupprimerCertificat

Cette fonction permet de supprimer un certificat installé sur machine[1]. Notons que ce procédé est irréversible c.-à-d. qu'elle doit être utilisée avec parcimonie. Le certificat supprimé sera celui pointé par la classe COutils lors de l'appel à SupprimerCertificat.

En-Tête : `SupprimerCertificat(LONG *lgResult);`

Pré-conditions :

- La classe COutils doit pointer vers un certificat.

[1] Les certificats installés dans la registry qui sont liés à une carte magnétique sont également supprimés de la base de registres. Toutefois, nous n'effaçons rien sur la carte. Dans l'état actuel des choses, nous ne pouvons pas encore effectuer des opérations directes sur les cartes magnétiques : nous devons passer par un logiciel spécifique. Notons que la suppression du certificat n'implique pas la suppression des clés qui lui sont associées.

Post-conditions :

- Le certificat pointé est supprimé de la liste des certificats.
- Le paramètre `lgResult` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement.

Détails d'implémentation :

```
STDMETHODIMP COutils::SupprimerCertificat(LONG *lgResult)
{
        *lgResult = DeleteCertificate();
        return S_OK;
}
```

2.2.8. ExporterCertificat

`ExporterCertificat` et `ImporterCertificat` ne sont pas des fonctions en lien direct avec le cahier de charges. Il s'agit d'offrir une nouvelle fonctionnalité : la possibilité de partager les clés publiques en distribuant les certificats[1].

Afin de garantir la confidentialité de ces échanges, les certificats seront chiffrés par des clés de chiffrement dérivées d'un mot de passe. De la sorte, un utilisateur fournissant un mot de passe (`password`) peut chiffré et stocké dans un fichier pour le transport ou le stockage le certificat sélectionné par `COutils`.

Nous pourrions envisager d'utiliser cette technique d'exportation pour protéger les clés privées. Si aucun certificat n'est installé sur la machine, personne ne peut signer puisque les clés privées sont choisies en fonction du certificat signataire.

En exportant ses certificats sur disquette et en les effaçant de son disque, Alice augmente le travail d'un fraudeur : celui-ci doit réinstaller tous les certificats avant de pouvoir utiliser les clés[2].

La méthode pour exporter un certificat est relativement simple :

1. Créer une structure spécifique pour le stockage du certificat dans un fichier,
2. Calculer le résumé du mot de passe pour sa protection,
3. Dériver une clé de session à partir du mot de passe,
4. Chiffrer et signer le certificat avec cette clé de session,
5. Ecrire le résultat dans un fichier.

En-Tête :
```
ExporterCertificat( LPSTR szStoreFile,          //out
                    LPSTR szPassword,           //in
                    LONG *lgResult);            //out
```

Pré-conditions :

- Un mot de passe doit être fourni.
- La classe `COutils` doit pointer vers un certificat.

Post-conditions :

- Le fichier `szStoreFile` contient le résultat du chiffrement du certificat par une clé de session dérivée du mot de passe `szPassword`.
- Le paramètre `lgResult` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement.

---

[1] Lors de l'exportation d'un certificat, ni la clé privée, ni la clé publique associées ne sont déplacées. Seul le certificat est exporté.

[2] Les clés d'Alice sont toujours installées puisque la fonction `SupprimerCertificat` n'efface aucune clé.

Détails d'implémentation :

```
STDMETHODIMP COutils::ExporterCertificat( LPSTR szStoreFile,
                                          LPSTR szPassword,
                                          LONG *lgResult)
{
      *lgResult = ExportCertificate(szStoreFile,szPassword);
      return S_OK;
}
```

2.2.9. ImporterCertificat

Cette fonction permet d'installer[1] un certificat sauvegardé dans un fichier (`szStoreFile`). A l'aide du mot de passe (`szPassword`), le contenu du fichier source sera déchiffré et le certificat sera installé sur la machine dans la branche `MY` de la `registry`.

En-Tête :
```
ImporterCertificat( LPSTR szStoreFile,          //in
                    LPSTR szPassword,           //in
                    LONG *lgResult);            //out
```

Pré-conditions :

- Le fichier `szStoreFile` doit exister.
- Le paramètre `szPassword` correspond au mot de passe à partir duquel le fichier `szStoreFile` a été chiffré.

Post-conditions :

- Le certificat se trouvant dans le fichier `szStoreFile` est installé sur la machine, dans le répertoire `MY` de la base des registres.
- Le paramètre `lgResult` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement.

Détails d'implémentation :

```
STDMETHODIMP COutils::ImporterCertificat( LPSTR szStoreFile,
                                          LPSTR szPassword,
                                          LONG *lgResult)
{
      *lgResult = ImportCertificate(szStoreFile,szPassword);
      return S_OK;
}
```

3. *Niveau 2 : Implémentation interne*

Si le premier niveau de notre analyse présente les services offerts par la classe `COutils` aux applications qui l'emploit, le second niveau se penche sur l'implémentation interne de chacune de ses méthodes internes. Par méthodes internes, nous entendons toutes les fonctions qui sont cachées à l'application. La raison pour laquelle ces méthodes internes sont cachées tient dans le fait qu'elles ne manipulent pas nécessairement des objets standards `COM` ou qu'elles ne correspondent pas à un service entier (dans le sens où certain service utilise d'autres méthodes internes).

[1] L'installation d'un certificat correspond à l'ajout d'une branche dans la base des registres correspondant au `system certificate store MY`. Aucune clé n'est installée.

Historiquement, c'est la structure interne (attributs et méthodes internes) que nous avons implémentée en premier lieu. Une fois cette structure terminée et testée, nous y avons ajouté une couche : la couche interface avec ses propres attributs et ses propres méthodes.

Les attributs et les méthodes internes à notre classe COutils sont représentés à la figure 6.3.

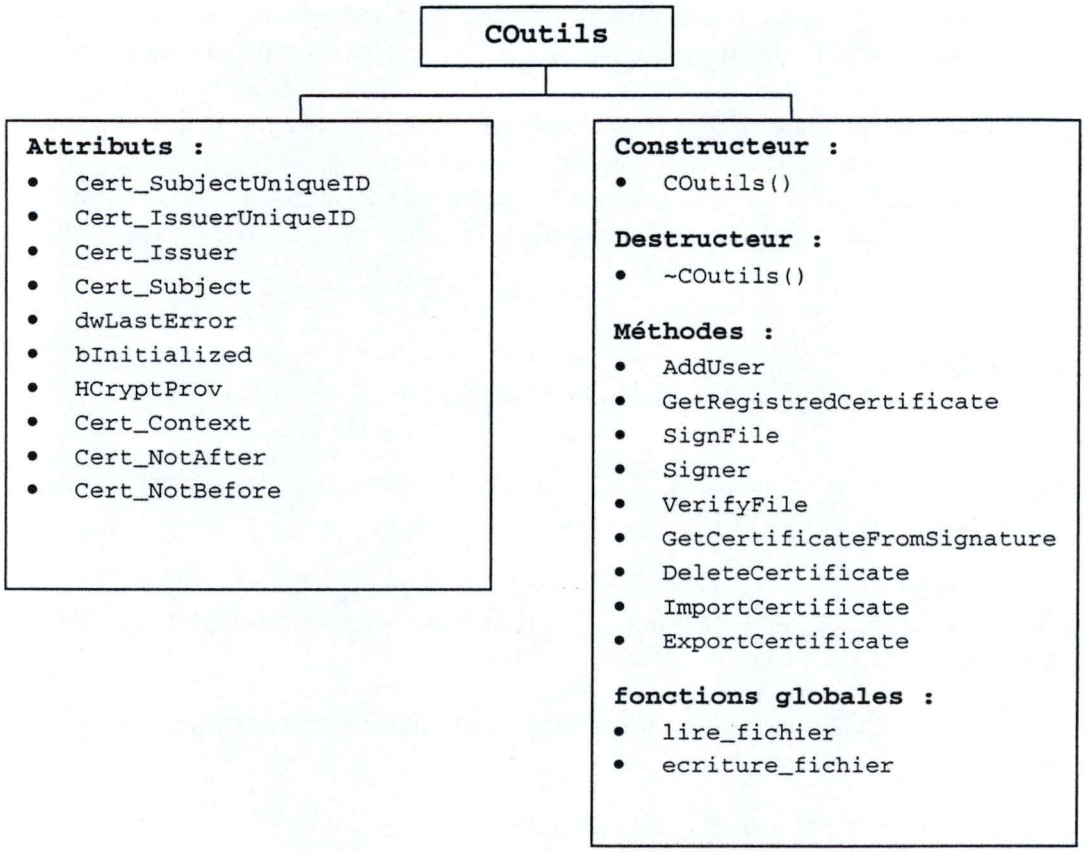

Figure 6.3. : structure interne de la classe COutils.

3.1. Smart Cards

Dans la description des cas concrets d'utilisation de la cryptoAPI, nous avons introduit le fait que la clé privée du signataire devait être utilisée lors de la signature. Pour cela, nous avions recherché son container de clé dans le CSP par défaut via la fonction CryptAcquireContext.

Dans le cahier de charges, nous avons indiqué que deux méthodes de stockage des clés devaient être considérées : le stockage dans la base des registres (registry) et le stockage sur cartes magnétiques (Smart Cards). Seul le premier cas a été traité jusqu'à present.

L'utilisation des cartes magnétiques est immédiate dès qu'on se rend compte que le module réalisant les opérations de cryptographie et mettant en jeu des cartes magnétiques est un CSP particulier qu'il suffit d'installer sur notre machine. La base de données de containers de clés de ce CSP indique si les clés sont stockées sur carte.

Pour les cartes magnétiques, nous utilisons le Gemplus CSP de la société GemSAFE. Ce CSP propose la signature numérique à l'aide de clés de type RSA stockées sur cartes GPK4000 Smart

Cards. Les clés et les certificats stockés sur les cartes sont chiffrés par un mot de passe (pincode) et la longueur des clés est limitée à 40 bits.

Lorsqu'un certificat stocké sur une Smart Card est installé sur une machine, il est installé dans la base de regitres et est référencé comme étant lié à une carte magnétique. Remarquons que cette installation ne copie pas la clé privée sur disque. De cette manière, quand une fonction de la cryptoAPI utilise un certificat lié à une carte magnétique pour une signature, elle fait appel au CSP de GemSafe pour accéder à la clé privée correspondante en lisant les informations sur la carte.

Le CSP de GemSAFE (la dll et le fichier de signature lui correspondant) est enregistré lors de l'installation du kit Smart Cards. Ainsi, toutes les fonctions de la cryptoAPI y ont accès. Lorsque nous tenterons d'obtenir un handle vers le container de clés associé à un certificat référencé comme étant lié à une carte magnétique, la fonction CryptAcquireContext indiquera que le CSP est le GemPlus CSP.

3.2. Attributs

Les attributs internes de la classe COutils sont de deux types : les attributs identifiant un certificat et les attributs de gestion de la cryptoAPI.

Nous avons déjà indiqué que la classe COutils proposait des fonctionnalités sur ou à partir d'un certificat. nous avons également vu que le certificat pointé par la classe était identifié par une application au travers des attributs de l'interface.

Ces attributs de l'interface[1] se retrouvent dans les attributs internes aux côtés d'autres attributs que nous définissons ci-dessous :

- LONG bInitialized;

Cet attribut détermine si un CSP a été acquis préalablement. Dans ce cas, sa valeur est 1.

- HCRYPTPROV HCryptProv;

Il s'agit du handle vers un CSP. Sa valeur est valide si bInitialized vaut 1.

- PCCERT_CONTEXT Cert_Context;

Cet attribut représente un pointeur vers une structure[2] contenant les informations qui caractérisent le certificat pointé par la classe.

- Cert_NotAfter & Cert_NotBefore :

Ces deux attributs déterminent les dates de validité du certificat pointé par la classe. Ces attributs ne se retrouvent pas dans les attributs d'interface puisqu'aucun type correspondant aux dates n'est compatible COM. Ces attributs sont utilisés dans l'implémentation des méthodes d'interface LireDateDebut et LireDateFin et sont de type FILETIME.

3.3. Constructeur et destructeur

Comme toutes les classes d'objets, la classe COutils possède son constructeur et son destructeur. Le rôle du premier sera d'initialiser les attributs et d'obtenir un handle vers le CSP par défaut tandis que le second libérera le handle vers le CSP acquis.

---

[1] Pour les définitions des attributs de l'interface, nous renvoyons le lecteur à la section 2.1. du présent chapitre.

[2] Comme pour bInitialized et HCryptProv, nous renvoyons le lecteur à la section 3.1. et 3.2. du chapitre V.

3.3.1. Constructeur

Ce constructeur est appelé lors de la création de chaque instance de la classe COutils. Il vise à obtenir un handle vers le CSP par défaut. Dans le cas où cette démarche échoue, il faut créer un nouveau container de clé pour l'utilisateur connecté. C'est le rôle de la fonction AddUser[1].

Nous écrirons pour le constructeur de la classe COutils :

```
//constructor of COutils
COutils()
{
LONG lgReturn;
this->dwLastError = (0);
this->bInitialized = (1);
this->Cert_Context = NULL;

//get the default CSP
lgReturn = (0);
lgReturn = ::CryptAcquireContext(
      &this->hCryptProv,
      NULL,                                    //default key container
      MS_DEF_PROV,                             //default CSP
      PROV_RSA_FULL,                           //type of default CSP
      0);
if (lgReturn != (1))                           //on failure
{
      this->dwLastError = GetLastError();
      this->bInitialized = (0);
      lgReturn = AddUser();                    //create a new container
}
//attributes initialisation
}
```

3.3.2. Destructeur

Il s'agit d'un appel à la fonction CryptReleaseContext sur l'attribut hCryptProv correspondant au CSP acquis par le constructeur ou lors d'un appel à une fonction auxiliaire.

```
//destructor of COutils
~COutils()
{
//release context provider if one acquired earlier
if (bInitialized == (1)) ::CryptReleaseContext(this->hCryptProv,0);
}
```

3.4. Méthodes internes

Dans cette section, nous présentons les spécifications des méthodes de la classe COutils, le but étant de montrer comment les outils de la cryptoAPI peuvent être utilisés pour effectuer des opérations de haut niveau.

Dans la section consacrée à l'implémentation de l'interface offerte par la classe COutils, nous avions indiqué que les méthodes de cette interface retournaient un paramètre d'erreur,

[1] Voir paragraphe 3.4.1. du présent chapitre.

lgResult. Les valeurs possibles de ce paramètre appartiennent à un type énuméré correspondant à tous les noms de fonctions de la cryptoAPI que notre classe utilise. Ainsi, par exemple :

```
0 = APIERROR,
1 = SUCCESS,
...
10 = CRYPTACQUIRECONTEXT,
11 = CERTOPENSYSTEMSTORE,
...
```

A l'utilisation de ce paramètre d'erreur, nous associons l'emploi de l'attribut DwLastError qui indique quel est le type de l'erreur produite par la fonction indiquée par lgResult. Par exemple, si lgResult vaut 10 et qu'à ce moment, l'attribut DwLastError vaut 4 (NTE_BAD_KEYSET), alors nous pouvons affirmer que nous avons tenté un appel à la fonction CryptAcquireContext sur une clé à laquelle aucun container de clés ne correspond.

A chaque appel à une fonction de la cryptoAPI, nous modifierons la valeur de l'attribut DwLastError et nous indiquerons, au travers de la valeur retournée par la méthode interne, le nom de la fonction fautive.

3.4.1. AddUser

La fonction AddUser permet de créer un container de clés.

Nous utiliserons la fonction CryptAcquireContext avec CRYPT_NEWKEYSET comme paramètre et le nom du nouveau container de clés correspond au nom de logon de l'utilisateur courant..

En-Tête : `LONG AddUser();`

Pré-conditions : aucune

Post-conditions :

- Un nouveau container de clé a été créé. Son nom correspond au nom de logon de l'utilisateur courant.
- Le paramètre lgReturnCode vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la cryptoAPI qui a provoqué l'erreur et l'attribut DwLastError nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::AddUser()
{
LONG   lgReturnCode = SUCCESS;
LONG   lgReturn;

//re-attempt to acquire a handle to the default key container.
lgReturn = (0);
lgReturn =::CryptAcquireContext(
       &hCryptProv,
       NULL,
       MS_DEF_PROV,
       PROV_RSA_FULL,
       0);
```

```
if (lgReturn !=(1))                    //on failure
{
        // Create default key container.
        if(!::CryptAcquireContext(
                &hCryptProv,
                NULL,
                MS_DEF_PROV,
                PROV_RSA_FULL,
                CRYPT_NEWKEYSET))
        {
                this->dwLastError = GetLastError();
                lgReturnCode = CRYPTACQUIRECONTEXT;
                return lgReturnCode;
        }
}
return lgReturnCode;                   //lgReturnCode is equal to SUCCESS
                                       //or equal to CRYPTACQUIRECONTEXT
}
```

3.4.2. GetRegistredCertificate

Un attribut interne à la classe (cela est transparent pour l'application) crée un lien physique vers le certificat (`Cert_Context`). Ce lien correspond à un pointeur vers la structure qui caractérise le certificat. Lors des différents appels aux fonctions de cryptographie, ce lien doit être utilisé pour spécifier le certificat à utiliser.

La fonction `GetRegistredCertificate` permet d'accéder à un certificat installé sur la machine[1]. Cet accès implique que la classe référence le certificat et que les attributs de la classe correspondent à ce dernier.

Pour obtenir les informations concernant les noms de la CA ou du propriétaire d'un certificat, nous utilisons la fonction `CertNameToStr` de la `cryptoAPI`. Selon le format d'encodage choisi, la chaîne de caractères renvoyée par `CertNameToStr` aura une forme différente. Nous employerons le format `CERT_X500_NAME_STR` qui correspond au classement des identifiants par type. Par exemple, si le nom du propriétaire du certificat vaut « `Michot Olivier` » et si son organisation se nomme « PBFLOW », alors la fonction `CertNameToStr` appelée avec le format `CERT_X500_NAME` nous donne pour le `Subject` :

« CN[2] = Michot Olivier, OU[3] = PBFlow »

A la suite d'un appel aux fonctions de lecture de certificats (`LireCertificatRegistry` et `LireCertificatSignature`), les attributs `Cert_Subject` et `Cert_Issuer` sont mis à jour. Si l'utilisateur désire consulter la valeur de ces attributs, il utilise les attributs de l'interface (qui convertissent les attributs internes dans un type compatible avec le objets COM). Deux possibilités s'offrent à lui :

- S'il désire n'obtenir que le nom du propriétaire du certificat, il écrira :
  `Name = MyClass.Cert_Subject(« CN »);`
  La variable `Name` de type `LPTSTR` contient dans ce cas la chaîne :
  « Michot Olivier »

[1] A partir d'ici, lorsque nous parlerons des certificats installés sur la machine, nous incluerons les certificats référencés comme étant liés à une carte magnétique.

[2] CN : Common Name

[3] OU : Organization Unit

- S'il désire obtenir toutes les informations connues sur le propriétaire, il écrira :
  `Name = MyClass.Cert_Subject(« FULL »);`
  La variable Name de type LPTSTR contient dans ce cas la chaîne :
  « Michot Olivier, PBFlow »

Ces deux méthodes d'accès aux attributs Cert_Subject et Cert_Issuer ont leur importance dans la mesure où nous ne connaissons pas systématiquement les identifiants qui déterminent le propriétaire et la CA. Ces identifiants varient en nombre et en type en fonction de l'autorité qui émet les certificats. Nous utiliserons la première méthode dans les cas où nous souhaitons accéder à un identifiant en particulier[1]. La seconde méthode sera appliquée si nous souhaitons connaître tous les indentifiants.

Méthode d'interface appelante : `LireCertificatRegistry`

En-Tête :
```
LONG GetRegistredCertificate( int nr,                    //in
                              LPSTR Store);              //in
```

Pré-conditions :

- Le paramètre `nr` indique le numéro du certificat à sélectionner (`nr>0`).
- `Store` indique dans quel `system store` le certificat doit être extrait.

Post-conditions :

- Si `nr` est inférieur au nombre de certificats installés dans le `system store` donné par `store`, la classe `COutils` pointe vers le certificat numéro `nr`. Les attributs sont mis à jour.
- Le paramètre `lgReturnCode` vaut `1` (SUCCESS) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la `cryptoAPI` qui a provoqué l'erreur et l'attribut `DwLastError` nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::GetRegistredCertificate(int nr,LPSTR Store)
{
LONG                            lgReturnCode = SUCCESS;

HCERTSTORE                      hCertStore = NULL;
PCCERT_CONTEXT                  pPrevCertContext = NULL;
PCCERT_CONTEXT                  pCertContext = NULL;

PCERT_NAME_VALUE                pvStructInfo = NULL;
DWORD*                          pcbStructInfo = NULL;

CERT_NAME_BLOB                  NameBlob;
DWORD                           dwSize;
LPSTR                           Attribute;
int                             i=1;

//return error if there is no selected CSP
if (!bInitialized)
{
      this->dwLastError = (0);
      lgReturnCode = NOTINITIALIZED;
      goto error;
}
```

[1] Si l'identifiant n'existe pas, un `String` (BSTR) vide sera renvoyé.

```
//Open the system certificate store
//we use here COutils.hCryptProv (acquire by the constructor) which
//is valid sinds bInitialized is not equal to 0.
hCertStore = CertOpenSystemStore(this->hCryptProv,Store);
if (!hCertStore)
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTOPENSYSTEMSTORE;
      goto error;
}

//read the certificate list
      for (; i<=nr;)
{
      pCertContext = CertEnumCertificatesInStore(hCertStore,
                                                 pPrevCertContext);
      if(!pCertContext)
      {
            this->dwLastError = GetLastError();
            lgReturnCode = CERTENUMCERTIFICATESINSTORE;
            goto error;
      }
      pPrevCertContext = pCertContext;
      i++;
}

if (!pCertContext) goto error;

//Search for the Subject
NameBlob = pCertContext->pCertInfo->Subject;
//get the size of Subject(NameBlob)
dwSize = ::CertNameToStr(
      MY_ENCODING_TYPE,
      &NameBlob,
      CERT_X500_NAME_STR,
      NULL,
      0);
if (dwSize<2)
{
      this->dwLastError = (0);
      lgReturnCode = CERTNAMETOSTR;
      goto error;
}
//allocate memory for Subject(Attribute is temporary)
if(!(Attribute = (LPSTR)malloc(dwSize)))
{
      this->dwLastError = (0);
      lgReturnCode = MALLOC;
      goto error;
}

      //store the Subject into Attribute
      dwSize = ::CertNameToStr(
      MY_ENCODING_TYPE,
      &NameBlob,
      CERT_X500_NAME_STR,
      Attribute,
      dwSize);
if(dwSize<2)
{
      this->dwLastError = (0);
      lgReturnCode = CERTNAMETOSTR;
      goto error;
}
//Cert_Subject is of X500 type
this->Cert_Subject = Attribute;
//we do the same for Cert_Issuer
//but we use NameBlob = pCertContext->pCertInfo->Issuer;
```

```
// Search for other informations concerning the selected certificate
//IssuerUniqueID
this->Cert_IssuerUniqueID =
                    pCertContext->pCertInfo->IssuerUniqueId.cbData;
//SubjectUniqueID
this->Cert_SubjectUniqueID =
                    pCertContext->pCertInfo->SubjectUniqueId.cbData;
//dates of validity
this->Cert_NotAfter = pCertContext->pCertInfo->NotAfter;
this->Cert_NotBefore = pCertContext->pCertInfo->NotBefore;
//the class needs to point to the choosen certificate
this->Cert_Context = pCertContext;

return lgReturnCode;

error :
return lgReturnCode;
}
```

3.4.3. SignFile

Cette fonction permet la signature de tous les types de données. Son principe se base sur la méthode de signature que nous avons exposé dans les cas concrets du chapitre V : à partir d'un nom de fichier source (`szSourceFile`) et du nom du fichier de signature (`szSignatureFile`), il s'agit de générer la signature des données du fichier source et de la placer, ainsi que le certificat et les identifiants des algorithmes de signature et de hash utilisés, dans le fichier de signature.

Notons que cette fonction utilise une autre fonction interne `Signer` qui réalise le véritable travail de signature. `SignFile` prend surtout en charge les entrées/sorties.

Méthode d'interface appelante : `SignerFichier`

En-Tête :
```
LONG SignFile( LPSTR szSourceFile,                    //in
               LPSTR szSignatureFile)                 //out
```

Pré-conditions :

- Le fichier à signer (`szSourceFile`) doit exister.
- La classe `COutils` doit pointer vers un certificat.

Post-conditions :

- Le fichier de signature (`szSignatureFile`) contient la signature du résumé de `szSourceFile`, le certificat pointé par la classe et les identifiants des algorithmes utilisé. Si le fichier de signature existe, son contenu est remplacé. Dans le cas contraire, il est créé.
- Le paramètre `lgReturnCode` vaut `1` (`SUCCESS`) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la `cryptoAPI` qui a provoqué l'erreur et l'attribut `DwLastError` nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::SignFile(LPSTR szSourceFile,LPSTR szSignatureFile)
{
LONG   lgReturnCode = SUCCESS;
LONG   lgReturn;

char   szContent[]= "blablabla";              //temporary
char*  pszContent = NULL;                     //temporary
```

```
BYTE* pbContent = (BYTE*) pszContent;          //message to be signed
DWORD cbContent = sizeof(szContent);           //size of message

BYTE* pbEncryptedBlob;                         //signed message
DWORD cbEncryptedBlob;                         //size of signed message

//return error if there is no selected CSP
if (!bInitialized)
{
      lgReturnCode = NOTINITIALIZED;
      goto error;
}

//read SourceFile and return his content (pbContent)
//and his size (cbContent)
lgReturn = (0);
lgReturn = lecture_fichier(szSourceFile,&pbContent,&cbContent);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

lgReturn = (0);
//signer is THE function used to sign ...
lgReturn = signer( true,
                   &pbContent,
                   &cbContent,
                   &pbEncryptedBlob,
                   &cbEncryptedBlob);

//the next error occurs when there is no key container
//OR WHEN THE KEYS ARE STORED ON SMART CARDS
//AND THERE IS NO SMART CARDS INSTALLED (unknown Gemplus CSP).
if (lgReturn == (10))
{
      lgReturnCode = 111;              //check for Smart Card !!!
      goto error;
}
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = SIGNER;
      goto error;
}

//write SignatureFile into szSignatureFile
lgReturn = ecriture_fichier( szSignatureFile,
                             pbEncryptedBlob,
                             cbEncryptedBlob);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

return lgReturnCode;

error:
return lgReturnCode;
}
```

La fonction Signer effectue exactement les mêmes opérations que l'exemple de signature traité dans la section 3.3. du chapitre V. Il s'agit de la sélection du container contenant la clé privée, du choix des paramètres de signature (algorithmes, ...) et de la signature proprement dite.

3.4.4. Signer

Méthode interne appelante : SignFile

En-Tête :
```
LONG Signer( BOOL detached,                    //in
             BYTE** pbContent,                 //in
             DWORD* cbContent,                 //in
             BYTE** pbEncryptedBlob,           //out
             DWORD* cbEncryptedBlob)           //out
```

Pré-conditions :

- detached indique si la signature doit être concaténée au message en entrée. Si detached vaut true, la signature est séparée.
- Les paramètres pbContent et cbContent sont respectivement les données à signer et la longueur de celles-ci.
- La classe COutils doit pointer vers un certificat.

Post-conditions :

- Les paramètres pbEncryptedBlob et cbEncryptedBlob sont respectivement la signature et la longueur de celle-ci.
- Le paramètre lgReturnCode vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la cryptoAPI qui a provoqué l'erreur et l'attribut DwLastError nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::Signer(BOOL detached,
                     BYTE** pbContent,
                     DWORD* cbContent,
                     BYTE** pbEncryptedBlob,
                     DWORD* cbEncryptedBlob)
{
LONG                        lgReturnCode = SUCCESS;
LONG                        lgReturn;

CRYPT_KEY_PROV_INFO*        pCryptKeyProvInfo;
DWORD                       cbData;
HCRYPTPROV                  hProv;

//create the MessageArray
const BYTE*    MessageArray[] = {*pbContent};

DWORD   MessageSizeArray[1];
MessageSizeArray[0] = *cbContent;

//get a pointer to your signature certificate
//we use the chosen certificate
PCCERT_CONTEXT      pSignerCert = NULL;
pSignerCert = this->Cert_Context;       //the selected certificate
```

```
//check if there is a private key associated with selected certificate
lgReturn = (0);
lgReturn = ::CertGetCertificateContextProperty(
        this->Cert_Context,
        CERT_KEY_PROV_INFO_PROP_ID,
        NULL,
        &cbData);
if (lgReturn != (1))
{
        this->dwLastError = GetLastError();
        lgReturnCode = CERTGETCERTIFICATECONTEXTPROPERTY;
        return lgReturnCode;
}

//allocate memory for properties..
pCryptKeyProvInfo = (CRYPT_KEY_PROV_INFO*)malloc(cbData);
if (!pCryptKeyProvInfo)
{
        this->dwLastError = (0);
        lgReturnCode = MALLOC;
        return lgReturnCode;
}

//get the properties of the selected Certificate.
lgReturn = (0);
lgReturn = ::CertGetCertificateContextProperty(
        this->Cert_Context,
        CERT_KEY_PROV_INFO_PROP_ID,
        pCryptKeyProvInfo,
        &cbData);
if (lgReturn != (1))
{
        this->dwLastError = GetLastError();
        lgReturnCode = CERTGETCERTIFICATECONTEXTPROPERTY;
        return lgReturnCode;
}

//get the container associated with the private key.
lgReturn = (0);
lgReturn = ::CryptAcquireContext(
        &hProv,
        pCryptKeyProvInfo->pwszContainerName,
        pCryptKeyProvInfo->pwszProvName,
        pCryptKeyProvInfo->dwProvType,
        pCryptKeyProvInfo->dwFlags);
if (lgReturn != (1))
{
        this->dwLastError = GetLastError();
        lgReturnCode = CRYPTACQUIRECONTEXT;
        return lgReturnCode;
}

//create a MsgCertArray
PCCERT_CONTEXT MsgCertArray[1];
MsgCertArray[0] = pSignerCert;

//Initialize the Algorithm Identifier structure
CRYPT_ALGORITHM_IDENTIFIER      HashAlgorithm;      //algorithm to hash
DWORD                           HashAlgSize;        //length of algorithm
HashAlgSize = sizeof(HashAlgorithm);                //get the size
memset(&HashAlgorithm, 0, HashAlgSize);             //allocate memory

//SHA is the chosen hash algorithme &
//RSA is used to generate the signature
HashAlgorithm.pszObjId = szOID_RSA_SHA1RSA;
```

```
//Initialize the signature structure
CRYPT_SIGN_MESSAGE_PARA   SigParams;          //signature parameters
DWORD                     SigParamsSize = sizeof(SigParams);
memset(&SigParams, 0, SigParamsSize);         //allocate memory

//set the structure values
SigParams.cbSize = SigParamsSize;                    //size
SigParams.dwMsgEncodingType = MY_ENCODING_TYPE;      //encoding types
SigParams.pSigningCert = pSignerCert;                //certificate
SigParams.HashAlgorithm = HashAlgorithm;             //hash algorithm
SigParams.cMsgCert = 1;
SigParams.rgpMsgCert = MsgCertArray;

//sign the message
BYTE**        pbEncryptedBlob;                       //signature
DWORD*        cbEncryptedBlob;                       //signature size
*cbEncryptedBlob = NULL;

//get the size of the output SignedBlob.
lgResult = CryptSignMessage(
      &SigParams,                        //signature parameters
      detached,                          //detached =true
      1,                                 //number of messages
      MessageArray,                      //messages to be signed
      MessageSizeArray,                  //size of messages
      NULL,                              //buffer for signed msg
      cbEncryptedBlob);                  //size of buffer

//allocate memory for the signed blob.
*pbEncryptedBlob = (BYTE*)malloc(*cbEncryptedBlob);
memset(*pbEncryptedBlob, 49, *cbEncryptedBlob);

//get the signature file
lgResult = CryptSignMessage(
      &SigParams,                        //signature parameters
      detached,                          //detached = true
      1,                                 //number of messages
      MessageArray,                      //messages to be signed
      MessageSizeArray,                  //size of messages
      *pbEncryptedBlob,                  //buffer for signed msg
      cbEncryptedBlob);                  //size of buffer
if(lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTSIGNMESSAGE;
      return lgReturnCode;
}

return lgReturnCode;
}
```

3.4.5. VerifyFile

La fonction de vérification est assez simple puisque la cryptoAPI réalise cette opération immédiatement. En effet, la fonction CryptVerifyDetachedMessageSignature permet de tester la validité d'une signature lorsqu'elle est contenue dans un fichier de signature.

Méthode de l'interface appelante : VerifierFichier

En-Tête :
```
LONG VerifyFile( LPSTR szSourceFile,                    //in
                 LPSTR szSignatureFile)                 //in
```

Pré-conditions :

- SzSourceFile est le fichier contenant le fichier en clair. Il doit exister.
- SzSignatureFile est le fichier de signature. Il doit exister.

Post-conditions :

- Le paramètre lgReturnCode vaut 1 (SUCCESS) si l'opération s'est déroulée correctement et la signature du fichier source peut être considérée comme valide. Dans le cas contraire, il indique le nom de la fonction de la cryptoAPI qui a provoqué l'erreur et l'attribut DwLastError nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::VerifyFile(LPSTR szSourceFile,LPSTR szSignatureFile)
{
LONG                lgReturn;
LONG                lgReturnCode = SUCCESS;

BYTE*               pbContent;                      //Signature
DWORD               cbContent;                      //size of Signature

CRYPT_VERIFY_MESSAGE_PARA VerifyParams;
DWORD               VerifyParamsSize = sizeof(VerifyParams);
char                VerifyArgs[] = "Arguments";

const BYTE*         rgpb[2];
DWORD               rgcb[2];

BYTE                *pbContenu;                     //plaintext
DWORD               cbContenu;                      //size of plaintext

memset(&VerifyParams,0,VerifyParamsSize);

//return error if there is no CSP
if (!bInitialized)
{
      this->dwLastError = (0);
      lgReturnCode = NOTINITIALIZED;
      goto error;
}

//read SignatureFile and return his content (pbContent)
//and his size (cbContent)
lgReturn = (0);
lgReturn = lecture_fichier(szSignatureFile,&pbContent,&cbContent);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

//prepare the verification parameters
VerifyParams.cbSize = VerifyParamsSize;
VerifyParams.dwMsgAndCertEncodingType = MY_ENCODING_TYPE;
VerifyParams.hCryptProv = hCryptProv;
VerifyParams.pfnGetSignerCertificate = NULL;
VerifyParams.pvGetArg = VerifyArgs;
```

```
//read SourceFile and return his content (pbContenu)
//and his size (cbContenu)
lgReturn = (0);
lgReturn = lecture_fichier(szSourceFile,&pbContenu,&cbContenu);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

rgpb[0] = pbContenu;
rgcb[0] = cbContenu;

//verify the signature
lgReturn = (0);
lgReturn = ::CryptVerifyDetachedMessageSignature(
      &VerifyParams,              //verification parameters
      0,                          //only one signer
      pbContent,                  //signature
      cbContent,                  //size of signature
      1,                          //number of array element in rgpb
      rgpb,                       //plaintext
      rgcb,                       //size of plaintext
      NULL);                      //not used here
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTVERIFYDETACHEDMESSAGESIGNATURE;
      goto error;
}

return lgReturnCode;

error:
return lgReturnCode;
}
```

3.4.6. GetCertificateFromSignature

Lorsque l'on désire accéder aux informations concernant le certificat contenu dans un fichier de signature, nous pouvons utiliser la fonction `GetCertificateFromSignature`. Cette fonction réalise les mêmes tâches que la fonction `GetRegistredCertificate` si ce n'est qu'elle s'applique à un fichier de signature dans lequel il n'y a qu'un seul certificat.

Méthode d'interface appelante : `LireCertificatSignature`

En-Tête : `LONG GetCertificateFromSignature(LPSTR szSignatureFile) //in`

Pré-conditions :

- `SzSignatureFile` est le fichier de signature. Il doit exister.

Post-conditions :

- la classe `COutils` pointe vers le certificat contenu dans le fichier `szSignatureFile`. Les attributs sont également mis à jour.
- Le paramètre `lgReturnCode` vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la `cryptoAPI` qui a provoqué l'erreur et l'attribut `DwLastError` nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::GetCertificateFromSignature(LPSTR szSignatureFile)
{
LONG                lgReturnCode = SUCCESS;
LONG                lgReturn;

BYTE*               pbContent;
DWORD               cbContent;

HCERTSTORE          hCertStore = NULL;
PCCERT_CONTEXT      pCertContext = NULL;

PCERT_NAME_VALUE    pvStructInfo = NULL;
DWORD*              pcbStructInfo = NULL;

CERT_NAME_BLOB      NameBlob;
DWORD               dwSize;
LPSTR               Attribute;

//read SignatureFile and return his content (pbContent)
//and his size (cbContent)
lgReturn = (0);
lgReturn = lecture_fichier(szSignatureFile,&pbContent,&cbContent);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

//get a handle to system store where the certificat is stored
hCertStore = ::CryptGetMessageCertificates(MY_ENCODING_TYPE,
                                           this->hCryptProv,
                                           0,
                                           pbContent,
                                           cbContent);
if (!hCertStore)
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTGETMESSAGECERTIFICATES;
      goto error;
}

//There is only one Certificate in SignatureFile
//we call CertEnumCertificateInStore once
pCertContext = ::CertEnumCertificatesInStore(hCertStore,pCertContext);
if(!pCertContext)
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTENUMCERTIFICATESINSTORE;
      goto error;
}

return lgReturnCode;

error:
return lgReturnCode;
}
```

3.4.7. DeleteCertificate

Cette méthode efface un certificat installé sur une machine[1]. La branche de la base des registres lui correspondant est supprimée. Rappelons qu'il s'agit d'une opération irréversible.

Fonction membre appelante : `SupprimerCertificat`

En-Tête : `LONG DeleteCertificate()`

Pré-conditions :

- La classe `COutils` doit pointer vers un certificat.

Post-conditions :

- Le paramètre `lgReturnCode` vaut `1` (`SUCCESS`) si l'opération s'est déroulée correctement et le certificat pointé par la classe `COutils` est supprimé de la base des registres. Dans le cas contraire, il indique le nom de la fonction de la `cryptoAPI` qui a provoqué l'erreur et l'attribut `DwLastError` nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::DeleteCertificate()
{
LONG   lgReturnCode = SUCCESS;
LONG   lgReturn;

//there must be a selected certificate
if (this->Cert_Context == NULL)
{
      lgReturn = APIERROR;
      return lgReturn;
}

//delete the certificate refered by Cert_Context
lgReturn = (0);
lgReturn = ::CertDeleteCertificateFromStore(this->Cert_Context);
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTDELETECERTIFICATEFROMSTORE;
      return lgReturnCode;
}

return lgReturnCode;
}
```

3.4.8. ExportCertificate

La méthode `ExportCertificate` n'a pas de lien direct avec les contraintes du cahier de charges, nous le savons. Elle permet aux utilisateurs de s'échanger leurs certificats et elle offre une possibilité suppémentaire pour la sécurisation des clés[2].

---

[1] Nous pouvons effacer des certificats liés à des cartes magnétiques. Notons que nous n'effaçons rien de la carte liée au certificat supprimé (que la carte soit dans le lecteur ou pas).

[2] Nous renvoyons le lecteur au paragraphe 2.2.8. du présent chapitre.

En plus, l'implémentation de cette méthode met bien en évidence la manière dont les appels aux fonctions de la cryptoAPI doivent être effectués. L'analyse de cette implémentation pourrait être d'un grand secours dans la mesure où c'est ce type de démarche qui devra être utilisé si nous voulons développer des fonctionnalités d'échanges de clés privées.

La démarche pour exporter un certificat dans un fichier (szStoreFile) est :

1. Créer une structure spécifique pour stockage du certificat dans un fichier
2. Calculer le résumé du mot de passe pour sa protection
3. Dériver une clé de session à partir du mot de passe
4. Chiffrer et signer le certificat avec cette clé de session
5. Ecrire le résultat dans un fichier

Méthode d'interface appelante : ExporterCertificat

En-Tête :
```
LONG ExportCertificate( LPSTR szStoreFile,          //out
                        LPSTR szPassword)           //in
```

Pré-conditions :

- La classe COutils doit pointer vers un certificat.
- Un mot de passe doit être fourni.

Post-conditions :

- Le contenu du fichier szStoreFile est remplacé par le certificat chiffré et signé. Si le fichier de destination n'existe pas, il est créé.
- Le paramètre lgReturnCode vaut 1 (SUCCESS) si l'opération s'est déroulée correctement. Dans le cas contraire, il indique le nom de la fonction de la cryptoAPI qui a provoqué l'erreur et l'attribut DwLastError nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::ExportCertificate(LPSTR szStoreFile,LPSTR szPassword)
{
LONG          lgReturnCode = SUCCESS;
LONG          lgReturn;

BYTE*         pbElement = NULL;          //serialized certificate
DWORD         cbElement;                 //size of serialized element

HCRYPTKEY     hKey;                      //key derived from the password
HCRYPTHASH    hHash;                     //hash

DWORD         dwLength;
DWORD         dwBufferLen = 0;

// return error if ther is no selected CSP
if (!bInitialized)
{
      this->dwLastError = (0);
      lgReturnCode = NOTINITIALIZED;
      goto error;
}
```

```
//get the size of the serialized element
lgReturn = (0);
lgReturn = ::CertSerializeCertificateStoreElement(
      this->Cert_Context,        //the selected certificate
      0,                         //not used, must be set to 0
      NULL,                      //to get the size
      &cbElement);               //size of serialized element
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTSERIALIZECERTIFICATESTOREELEMENT;
      goto error;
}

//allocate memory for the serialized element.
pbElement = (BYTE*)malloc(cbElement);
if (!pbElement)
{
      this->dwLastError = (0);
      lgReturnCode = MALLOC;
      goto error;
}

//get the serialized element from the selected certificate
lgReturn = (0);
lgReturn = ::CertSerializeCertificateStoreElement(
      this->Cert_Context,        //the selected certificate
      0,                         //not used, must be set to 0
      pbElement,                 //serialized element
      &cbElement);               //size of serialized element
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTSERIALIZECERTIFICATESTOREELEMENT;
      goto error;
}

//create an empty hash object.
lgReturn = (0);
lgReturn = ::CryptCreateHash(
      hCryptProv,                      //CSP to use
      CALG_MD5,                        //algorithme to use
      0,                               //no key
      0,                               //not used here
      &hHash);                         //the empty hash
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTCREATEHASH;
      goto error;
}

dwLength = strlen(szPassword);

//hash the password.
lgReturn = (0);
lgReturn = ::CryptHashData(
      hHash,                           //the hashed password
      (BYTE *)szPassword,              //password
      dwLength,                        //length of password
      0);                              //ignored by default CSP
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTHASHDATA;
      goto error;
}
```

```
//create a session key based on the hash of the password.
//the key is of CRYPT_EXPORTABLE type which means that the session key
//can be transfered out of the CSP.
lgReturn = (0);
lgReturn = ::CryptDeriveKey(
      hCryptProv,                    //CSP to use
      CALG_RC4,                      //Algorithm to use
      hHash,                         //hashed password
      CRYPT_EXPORTABLE,              //default type
      &hKey);                        //the derived key
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTDERIVEKEY;
      goto error;
}

//encrypt the Serialized Certificate with our derived key
//get the size of the Buffer
lgReturn = (0);
lgReturn = ::CryptEncrypt(
      hKey,                          //bderived from the password key
      hHash,                         //sign and encrypt
      true,                          //one block to be encrypted
      0,                             //not used here
      NULL,                          //get the size
      &pbElement,                    //encrypted certificate
      dwBufferLen);                  //size of the encrypted cert.
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTENCRYPT;
      goto error;
}

//encrypt the Serialized Certificate with our derived key
lgReturn = (0);
lgReturn = ::CryptEncrypt(
      hKey,                          //derived from the password key
      hHash,                         //sign and encrypt
      true,                          //one block to be encrypted
      0,                             //not used here
      pbElement,                     //get the size
      &pbElement,                    //encrypted certificate
      dwBufferLen);                  //size of the encrypted cert.
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTENCRYPT;
      goto error;
}

//write the encrypted serialized certificate into szStoreFile
lgReturn = (0);
lgReturn = ecriture_fichier(szStoreFile,pbElement,cbElement);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

return lgReturnCode;

error:
return lgReturnCode;
}
```

3.4.9. ImportCertificate

Cette méthode fait exactement l'inverse de la méthode ExportCertificate.

A partir d'un mot de passe (szPassword), elle déchiffre un certificat se trouvant dans un fichier (szStoreFile) et elle l'installe dans la branche MY de la base de registres. Dans le cas où le mot de passe n'est pas celui qui a permis le chiffrement, l'opération échoue puisque le contenu du fichier n'est pas déchiffré correctement. Si le certificat a été modifié, une erreur est produite puisque le résumé du fichier ne correspond plus.

Fonction membre appelante : ImporterCertificat

En-Tête :
```
LONG ImportCertificate( LPSTR szStoreFile,          //in
                        LPSTR szPassword)           //in
```

Pré-conditions :

- Un mot de passe doit être fourni.
- Le fichier contenant le certificat à importer doit exister.

Post-conditions :

- Le paramètre lgReturnCode vaut 1 (SUCCESS) si l'opération s'est déroulée correctement et le certificat contenu dans le fichier szStoreFile est installé[1] sur la machine dans le system store « MY ». Dans le cas contraire, il indique le nom de la fonction de la cryptoAPI qui a provoqué l'erreur et l'attribut DwLastError nous informe sur le type d'erreur rencontrée.

Détails d'implémentation :

```
LONG COutils::ImportCertificate(LPSTR szStoreFile,LPSTR szPassword)
{
LONG                lgReturnCode = SUCCESS;
LONG                lgReturn;

BYTE*               pbElement = NULL;          //serialized certificate
DWORD               cbElement;                 //size of pbElement

HCRYPTKEY           hKey;                      //derived key
HCRYPTHASH          hHash;                     //hash

PCCERT_CONTEXT      pCertContext = NULL;       //new certificate
HCERTSTORE          hSystemStore;              //system store

DWORD               dwLength;
DWORD               dwBufferLen = 0;

// return error if there is no selected CSP
if (!bInitialized)
{
      this->dwLastError = (0);
      lgReturnCode = NOTINITIALIZED;
      goto error;
}
```

[1] Si le certificat existe déjà dans la base des registres, aucun remplacement n'est effectué.

```
//open the MY system certificate store.
hSystemStore = ::CertOpenSystemStore(this->hCryptProv,"MY");
if (!hSystemStore)
{
      this->dwLastError = GetLastError();
      lgReturnCode = CERTOPENSYSTEMSTORE;
      goto error;
}

//read the StoreFile and return his content (pbElement)
//and his size (cbElement)
lgReturn = (0);
lgReturn = lecture_fichier(szStoreFile,&pbElement,&cbElement);
if (lgReturn != (1))
{
      this->dwLastError = (0);
      lgReturnCode = FICHIER;
      goto error;
}

//create an empty hash object.
lgReturn = (0);
lgReturn = ::CryptCreateHash(
      hCryptProv,                    //CSP to use
      CALG_MD5,                      //algorithme to use
      0,                             //no key
      0,                             //not used here
      &hHash);                       //the empty hash
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTCREATEHASH;
      goto error;
}

dwLength = strlen(szPassword);
//hash the password.
lgReturn = (0);
lgReturn = ::CryptHashData(
      hHash,                         //the hashed password
      (BYTE *)szPassword,            //password
      dwLength,                      //length of password
      0);                            //ignored by default CSP
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTHASHDATA;
      goto error;
}

//create a session key based on the hash of the password.
//the key is of CRYPT_EXPORTABLE type which means that the session key
//can be transfered out of the CSP.
lgReturn = (0);
lgReturn = ::CryptDeriveKey(
      hCryptProv,                    //CSP to use
      CALG_RC4,                      //Algorithm to use
      hHash,                         //hashed password
      CRYPT_EXPORTABLE,              //default type
      &hKey);                        //the derived key
if (lgReturn != (1))
{
      this->dwLastError = GetLastError();
      lgReturnCode = CRYPTDERIVEKEY;
      goto error;
}
```

```
//decrypt the Serialized Certificate
if(!CryptDecrypt(
        hKey,                           //derived from the password key
        hHash,                          //signed and encrypted
        true,                           //one block to be decrypted
        0,                              //not used here
        pbElement,                      //encrypted certificate
        &cbElement))                    //get the size
{
        this->dwLastError = GetLastError();
        lgReturnCode = CRYPTENCRYPT;
        goto error;
}

//add the new certificate to SystemStore.
lgReturn = (0);
lgReturn = ::CertAddSerializedElementToStore(
        hSystemStore,                   //MY system store
        pbElement,                      //serialized element
        cbElement,                      //length of pbElement.
        CERT_STORE_ADD_USE_EXISTING,    //do not replace
        0,                              //not used here
        CERT_STORE_CERTIFICATE_CONTEXT_FLAG,    //add a certificate
        NULL,                           //optional
        NULL);                          //optional
if (lgReturn != (1))
{
        this->dwLastError = GetLastError();
        lgReturnCode = CERTADDSERIALIZEDELEMENTTOSTORE;
        goto error;
}

return lgReturnCode;

error:
return lgReturnCode;
}
```

Chapitre VIII : Scénarios d'utilisation

Tout au long des chapitres précédents, nous avons construit des outils de signature numérique. Dans ce chapitre, nous présenterons comment nous avons utilisés concrètement ces outils et cela dans deux types différents d'applications. Il s'agit de la phase de déploiement de notre projet.

D'après le cahier de charges, nous devions acquérir ou concevoir des solutions de signature numérique capables d'être utilisées par un logiciel de signature indépendant de l'application `PBFlow` et par cette application elle-même.

A l'image de ces deux utilisations, nous construirons notre chapitre selon deux axes qui traiterons respectivement :

- de l'intégration de la classe `COutils` dans un logiciel développé en `Visual Basic`
- de la création d'un composant de signature téléchargeable au travers de l'*Internet*.

Au travers de ce chapitre, nous justifierons nos choix de développement en présentant les résultats atteints et en démontrant que ceux-ci recouvrent entièrement le cahier de charges.

Nous ne nous intéresserons pas ici à l'implémentation pratique. Nous nous concentrerons sur les appels faits à la classe `COutils` et sur les fonctionnalités de cryptographie offertes aux utilisateurs.

1. *Logiciel de signature numérique*

Un des tous premiers objectifs de notre projet était de réaliser un logiciel de signature (indépendamment de l'application `PBFlow`) qui offrirait l'ensemble des fonctions de cryptographie et de gestion des certificats définies dans le cahier de charges.

La partie Interface Homme-Machine de ce logiciel a été développée en `Visual Basic` par M. Bernard Ramaekers. Il s'est occupé de toute la gestion des écrans mais également de la gestion des différents modules de l'architecture. Notre rôle a été d'intégrer la classe `COutils` et d'utiliser celle-ci pour réaliser les tâches de cryptographie et de gestion des certificats.

1.1. Initialisation d'une instance de la classe Coutils

Au démarrage du logiciel de signature, une instance de la classe `COutils` est créée. Cette création correspond, en `Visual Basic` (VB) à la ligne suivante :

```
Public objCrypto As New SECURITELib.Outils
```

Notre `dll` de signature se nomme `securite.dll`. Elle est considérée par VB comme une librairie n'offrant qu'un seul et unique outils :la classe `COutils` dont le constructeur est `Outils()`.

Cet outil nouvellement créé (`objCrypto`) n'offre aux utilisateurs que les fonctionnalités prévues par le cahier de charges[1] : il s'agit de l'interface entre la classe et l'application. Ces fonctionnalités représentent les attributs et les méthodes de la classe définissant l'interface et sont accessibles sous la forme d'une `properties list` que nous présentons à la figure 8.1. :

Figure 8.1. : liste des propriétés de l'objet `objCrypto`.

Rappelons que l'utilisation du constructeur de la classe correspond à la connexion au container de clés par défaut ou à la création d'un nouveau container dans le cas où il n'en existerait aucun.

Nous sommes fins prêts : « Que le spectacle commence ! ».

Les trois grandes fonctions qu'offre le logiciel (la signature, la vérification et la gestion des certificats) sont présentées sous formes d'icônes. La figure 8.2. correspond à l'écran principal du logiciel.

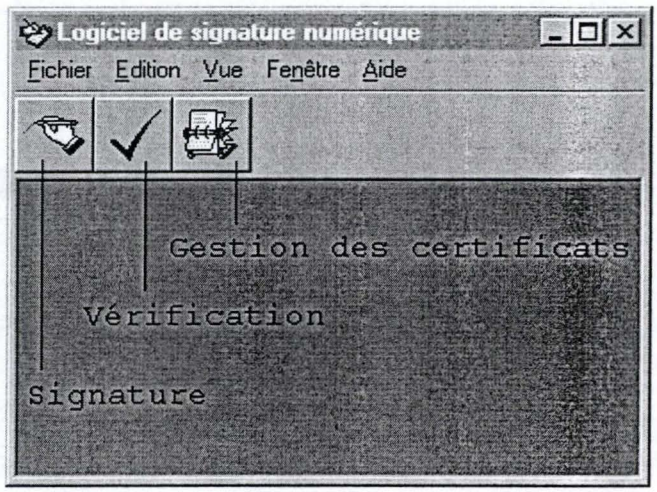

Figure 8.2. : logiciel de signature.

Chacune de ces trois fonctionnalités sera traitée dans une des sections suivantes. Dans ces sections, nous ne commenterons pas les choix relatifs à la présentation des écrans, nous montrerons simplement comment la classe `COutils` a été intégrée et utilisée par le logiciel. Il s'agira de mettre en évidence les appels à la classe et de justifier, par des exemples concrets, nos choix de développement.

[1] Ces fonctionnalités sont définies dans l'implémentation de la classe `COutils` (chapitre `VI`) à la section 2.

1.2. Signature numérique

Toutes les opérations de signature s'effectuent à partir de l'écran présenté à la figure 8.3.

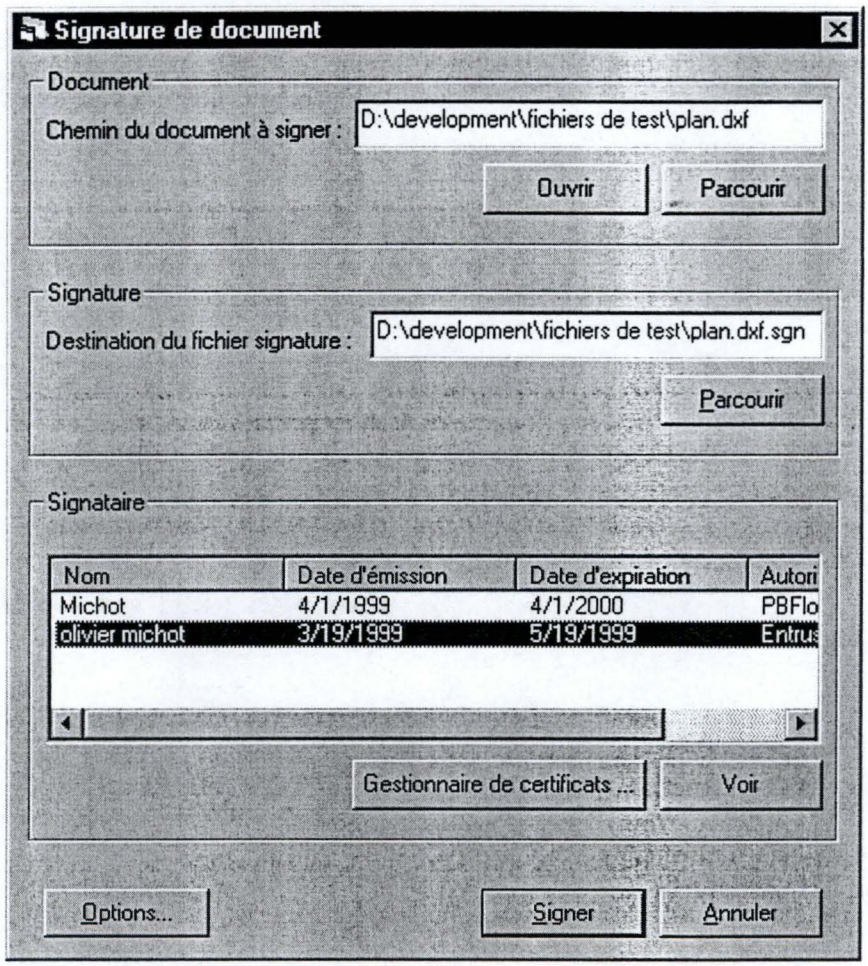

Figure 8.3. : signature d'un document

Pour signer un document, il suffit d'utiliser la méthode de l'interface `SignerFichier`. L'utilisateur doit choisir un fichier source (champs `Document`) et un fichier de signature (champs `Signature`). Il doit également sélectionner un certificat dans la liste affichée.

Lorsque tous les arguments utiles à l'appel de la fonction `SignerFichier` sont connus, nous pouvons écrire :

```
//get the selected certificate
//lsvMesCertificats.SelectedItem.Index is index of the certificat in the list
//MY is the system store
//lgResult corresponds to the error parameter
objCrypto.LireCertificatRegistry lsvMesCertificats.SelectedItem.Index,
                                 "MY",
                                 lgResult
```

```
If (lgResult = 1) Then

        //SignerFichier txtDocument and put signature in txtSignature
        objCrypto.SignerFichier txtDocument.Text, txtSignature.Text, lgResult
        If (lgResult = 1) Then
        MsgBox  "Le fichier a été correctement signé.",vbInformation+vbOKOnly,
                "Information"
        End If
End If
```

La procédure de signature est donc très simple, car elle ne nécessite pas de grandes manipulations de la part de l'utilisateur. Les contraintes du cahier de charges sont tout à fait respectées et les options de signature sont accessibles via le bouton `Options` (figure 8.4).

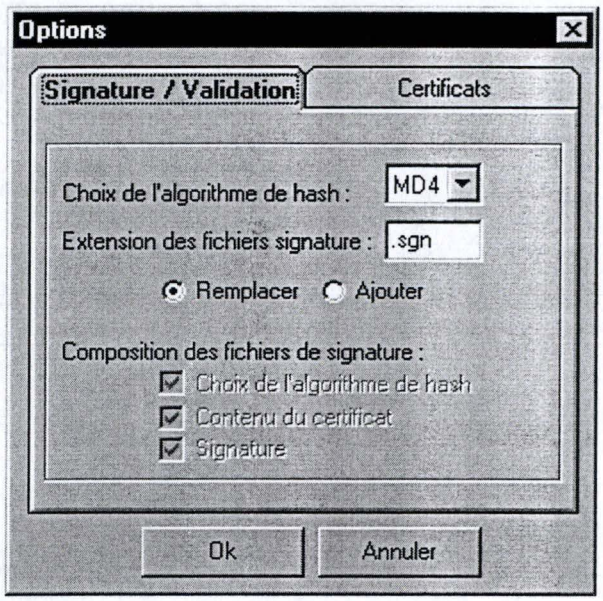

figure 8.4. : choix des options de signature.

Un écran de confirmation permet de rappeler à l'utilisateur qu'il va signer un document. Cela permet d'éviter les erreurs de manipulation (figure 8.5.) :

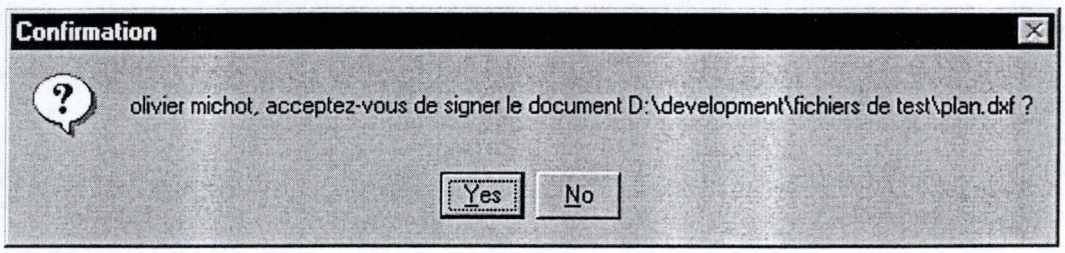

Figure 8.5. : confirmation de la signature.

Puisque la procédure de signature nécessite l'emploi de la clé privée du signataire, la méthode `SignerFichier` doit accéder au container de clé. Si le certificat du signataire est référencé comme étant lié à une carte magnétique, ce container contenant la clé privée se trouve sur une carte magnétique. Il faudra donc aller lire sur cette carte.

La lecture sur la carte magnétique est immédiate et ne demande aucune manipulation supplémentaire si ce n'est l'introduction d'un `PINCode` (figure 8.6.).

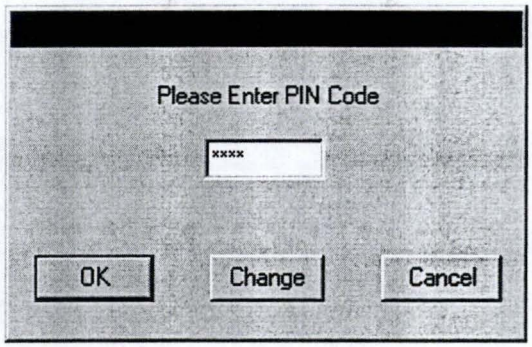

Figure 8.6. : introduction du `PINCode`.

Que la clé soit sur une carte ou dans la base des registres, l'application doit contacter le container de clés (via le système d'exploitation). L'`OS` indique alors à l'utilisateur qu'une application tente d'accéder à son container et que cela requiert son autorisation (figure 8.7.).

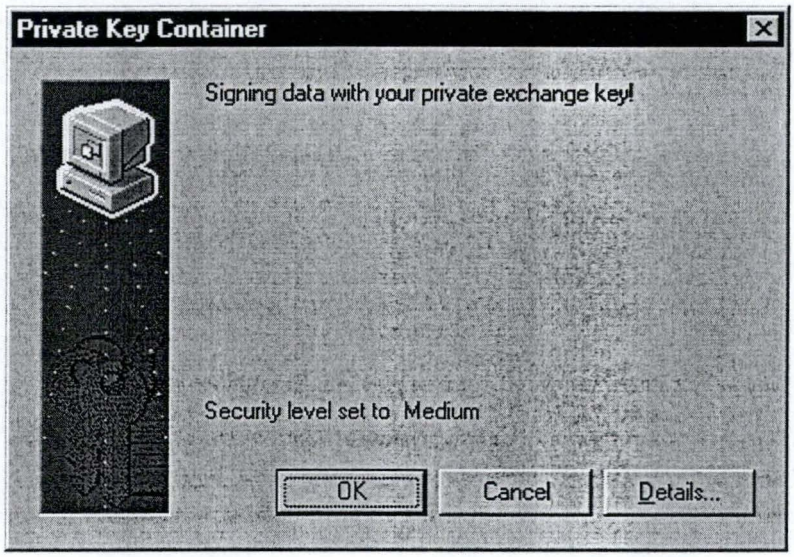

Figure 8.7. : demande d'accès au container de la clé privée du signataire.

Lorsque l'opération de signature s'est déroulée normalement, le fichier de signature est créé et nous en informons l'utilisateur (figure 8.8.).

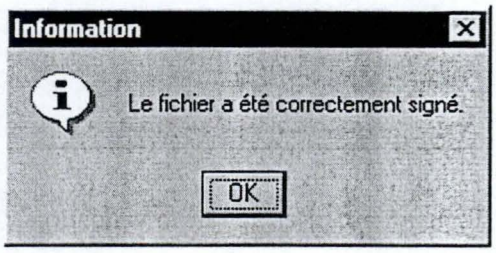

Figure 8.8. : résultat de la signature.

1.3. Vérification de signature

Les données nécessaires à la vérification sont le document en clair et le fichier de signature correspondant (figure 8.9.).

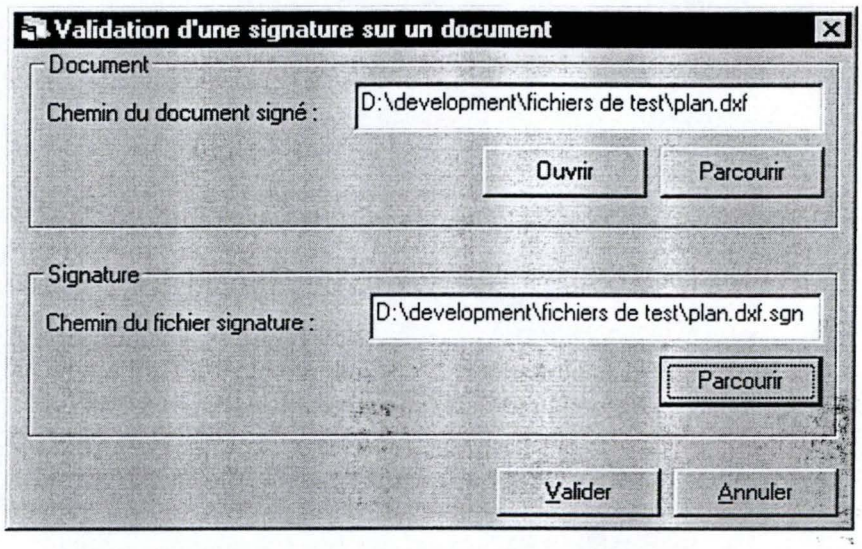

Figure 8.9. : vérification d'une signature.

Les chemins d'accès aux deux fichiers sont passés en arguments à `VerifierFichier` :

```
//VerifierFichier Document with signature in Signature
objCrypto.VerifierFichier frmValidation.txtDocument.Text,
                          frmValidation.txtSignature.Text,
                          lgResult
If (lgResult = 1) Then
      //get the certificate from the signature File
      objCrypto.LireCertificatSignature frmValidation.txtSignature.Text,
                                        lgResult
End if
```

Dans le cas où la signature correspond au fichier en clair, l'objet `objCrypto` pointe vers le certificat stocké dans le fichier de signature et les résultats sont affichés (figure 8.10.).

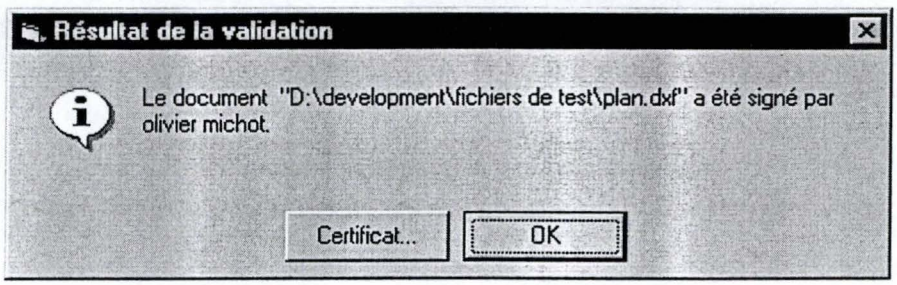

Figure 8.10. : Résultat de la validation.

1.4. Gestionnaire de certificats

La troisième fonctionnalité proposée par le logiciel est le gestionnaire de certificats. Son rôle est de visualiser, demander, vérifier et effacer un certificat installé sur la machine. Il offre également la possibilité de visionner les certificats des autorités de certification que l'utilisateur emploie. La figure 8.11. présente le gestionnaire de certificats.

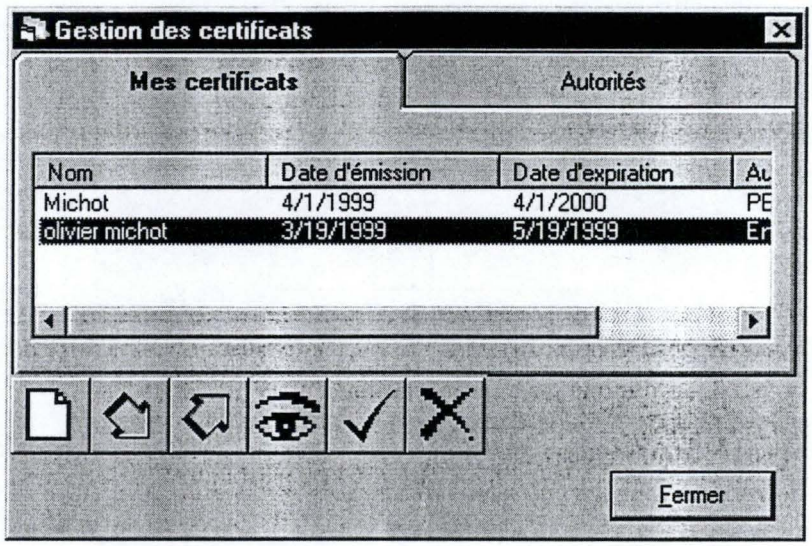

Figure 8.11. : gestionnaire de certificats.

Les informations stockées dans un certificat sont affichées à l'aide de l'écran suivant (figure 8.12.). Cette fonction de visualisation est accessible à partir des deux autres grandes fonctionnalités.

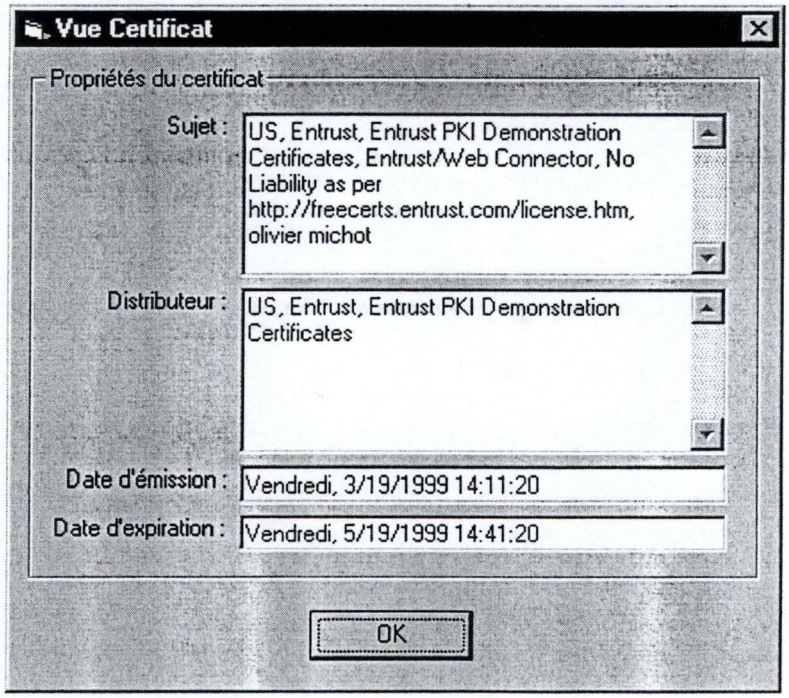

Figure 8.12 : visualisation du contenu d'un certificat

La liste des certificats affichée par le gestionnaire de certificats correspond au parcours des branches de la base des registres à l'aide de la fonction `LireCertificatRegistry`. Les informations telles que les noms du sujet ou de la CA sont obtenues via les attributs de l'objet `objCrypto`. Les dates de validité sont extraites grâce à la fonction `LireDateDebut` et `LireDateFin`.

L'importation et l'exportation des certificats se fait grâce au chiffrement/déchiffrement du certificat par une clé de session dérivée d'un mot de passe (figure 8.13.).

Figure 8.13. : importation d'un certificat.

La suppression d'un certificat étant une opération irréversible, nous demandons systématiquement la confirmation à l'utilisateur (figure 8.14.).

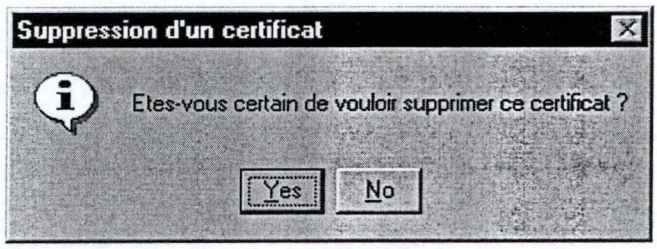

Figure 8.14. : confirmation de la suppression d'un certificat.

Quant aux deux dernières fonctions offertes par le gestionnaire de certificats (vérification et requête), elles permettent à l'utilisateur de se connecter directement sur le site de la CA.

2. *Composant de signature numérique*

Lorsqu'un utilisateur de PBFlow veut ajouter un document à un dossier électronique, il a la possibilité de le signer au moyen de la fonction de signature proposée par l'application PBFlow (figure 8.15).

Ce sera donc cette application qui réalisera les opérations de signature, à savoir :

- la sélection du certificat du signataire,
- la génération d'un fichier de signature.

Ces deux opérations nécessitent chaque fois un appel à notre classe COutils.

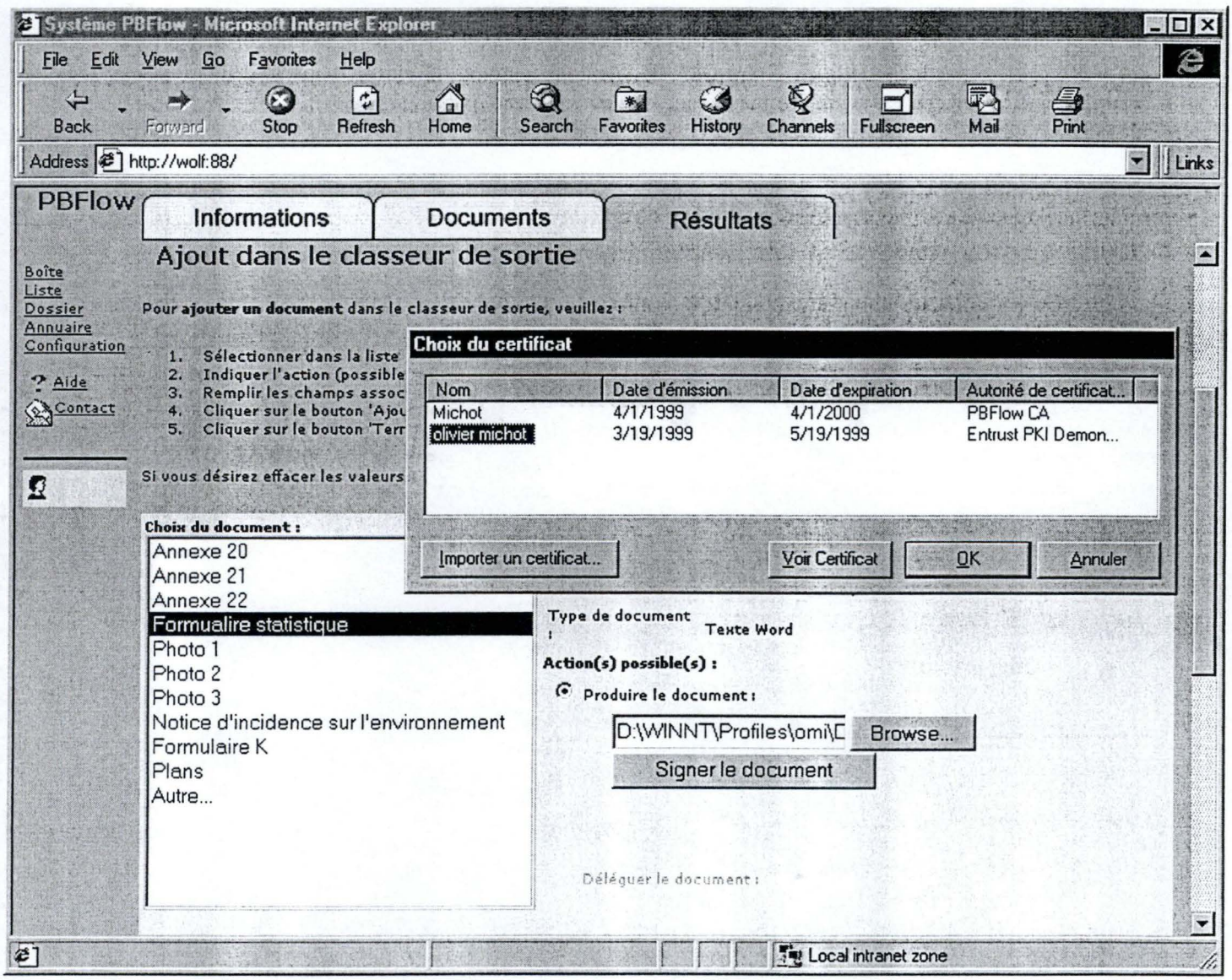

Figure 8.15. : signature dans PBFlow.

Pour des raisons de sécurité et de performance, nous avons décidé que les fonctions de signature seraient effectuées du côté « client » c.-à-d. du côté de l'utilisateur connecté à l'application PBFlow au travers de l'*Internet*.

En effet, si les fonctions de cryptographie devaient être réalisées du côté « serveur », il faudrait transmettre à celui-ci (au travers du réseau) toutes les informations utiles à la signature, en particulier les clés privées ! De plus, dans le cas où la signature échouerait, nous avons estimé que si l'utilisateur devait attendre un message d'erreur du « serveur », le temps de réponse de l'application ne serait plus acceptable.

L'application PBFlow utilise un composant de signature (développé par nos soins) qui sera installé chez le « client » lors de l'ouverture de la connexion. C'est ce composant réalisera les opérations de signature sur la machine du « client ». Le fait que la classe COutils soit encapsulée dans une dll et qu'elle utilise principalement des objets compatibles COM, facilite grandement le développement d'un tel composant appelé OCX.

Un ActiveX control (OCX) est un composant exécutable qui peut être ajouté à une page WEB et appelé à partir de celle-ci. Notre OCX de signature emploie la classe COutils de la même manière que le logiciel de signature présenté à la section précédente que les certificats soient stockés dans la base des registres ou dans des cartes magnétiques.

En plus, du choix du certificat du signataire et de la signature proprement dite, nous avons inclu une fonction d'importation de sorte que l'utilisateur qui aurait sauvegardé ses certificats sur disquette puisse utiliser ceux-ci sans devoir lancer notre logiciel de signature.

Ce dernier est toujours très utile dans la mesure où l'application PBFlow n'a pas pour but de proposer toutes les focntions de cryptographie et de gestion de certificats.

La figure 8.15. nous a présenté une utilisation typique de l'OCX de signature par l'application PBFlow. Lorsque l'utilisateur consulte sa boîte de travail, il peut ajouter un document au classeur de sortie (ajout au dossier). Lorsqu'il a sélectionné le document qu'il veut ajouter, il peut choisir de le signer d'une simple pression sur le bouton « Signer le document ». Cette action appelle l'OCX qui prend la main et la conserve jusqu'à ce que les opérations de signature soient terminées (réussite ou échec).

Conclusions

A la fin de ce mémoire, il serait utile de rappeler les objectifs que nous nous étions fixés. Il s'agissait d'étudier des techniques de signature numérique et de les mettre en œuvre dans le cadre du projet `PBFlow`.

Dans un premier temps, nous avons introduit les éléments théoriques utilisés dans ce travail. Ensuite, nous avons présenté la manière dont nous avons géré notre projet d'informatisation en décrivant successivement les phases de préparation, de développement et de déploiement.

Dans la phase de préparation, nous avons défini quatre étapes : l'identification des tenants et des aboutissants du projet `PBFlow`, la rédaction du cahier de charges imposées, l'évaluation des possibilités du marché et finalement, la présentation de nos choix de développement qui comprenaient la classe `COutils` et la `cryptoAPI`.

Dans la phase de développement, nous avons analysé chacune des fonctionnalités implémentées. Cette analyse a été réalisée sur deux niveaux afin de bien séparer l'implémentation interne des outils de signature et l'implémentation des services offerts aux applications faisant appel à notre classe.

Dans la phase de déploiement, nous avons mis en évidence la manière dont les outils que nous avons développés sont utilisés dans des applications concrètes et en particulier dans l'application `PBFlow`.

En conclusion, nous pouvons dire que les résultats atteints au terme de ce travail sont très satisfaisants puisqu'ils couvrent tous les aspects du cahier de charges.

Cependant, il ne faudrait pas s'endormir sur ses lauriers, il reste encore beaucoup de travail : en effet, le logiciel de signature n'offre qu'un certain nombre de fonctionnalités qu'il faudrait augmenter. Si le logiciel est utilisé pour les échanges de documents en dehors de l'environnement `PBFlow`, il serait intéressant d'ajouter le chiffrement et le déchiffrement de documents. De plus, nous ne sommes pas parvenus (dans les délais fixés) à distribuer des certificats et leur clé privée.

Nous avons parlé de l'utilisation de la carte magnétique. Une amélioration utile serait de rendre possible l'installation de certificats sur les cartes sans passer par le logiciel distribué par le constructeur. Si notre logiciel de signature pouvait prendre cette fonctionnalité en charge, cela permettrait de rassembler toutes les opérations de gestion des certificats dans un même module ...

Je terminerai ce mémoire en exprimant le plaisir que j'ai eu à travailler sur un sujet aussi passionnant et je remercie encore une fois toutes les personnes qui me l'ont permis.

Références

1. *Logiciels de signature testés*

[1] Point 'n Sign (http://www.soundcode.com/)
[2] JDSS II (http://www.digitalkey.com/)
[3] Entrust/Entellignence (http://www.entrust.com/)
[4] SafenSign (http://www.securisys.com/safensigned.htm)
[5] SafeGuard Sign&Crypt 2.0 (http://www.e-lock.com/)
[6] Regnoc Software (http://www.regnoc.com/)
[7] cryptlib (http://www.cs.auckland.ac.nz/~pgut001/cryptlib/)
[8] E-Lock ATS (http://www.e-lock.com/)

2. *Informations sur les certificats*

[1] Techiques and methods for Network Security - X.509 Certificate Format
http://www.net-quest.com/~jy/term_paper/x509.html
[2] OII - Information Security Standards
http://www.echo.lu/oii/en/secure.html
[3] OII - Information Security Standards
http://www.echo.lu/oii/en/directory.html
[4] Certificate Services
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/certsrv/crtsv_about_7jn7.htm

3. *Autorités de certification*

[1] VeriSign : Repository Introduction to Client Digital IDs^{SM}
http://www.verisign.com/repository/brwidint.html
[2] BelSign, The Digital Certification Authority for Europe
http://www.belsign.be
[3] GlobalSign, Proposal for a beneficial collaboration between FUNDP and GlobalSign,09/02/1999

4. *CryptoAPI*

[1] The crypapi Component :
http://msdn.microsoft.com/library/tools/wcesdkr/modc7sok.htm
[2] The Cryptography API, or How to Keep a Secret :
http://msdn.microsoft.com/library/techart/msdn_cryptapi.htm
[3] Microsoft CryptoAPI Overview
http://www.microsoft.com/workshop/c-frame.htm#/workshop/security/default.asp
[4] Microsoft CryptoAPI 2.0 : Contents of the CryptoAPI Documentation
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/crypto/0guide_4vqo.htm

[5] The CryptoAPI Programming Model
http://www.inch.com/~tang/sources/crypto/crypto_tutorial.html
[6] Supporting CryptoAPI in Real-World Applications
http://msdn.microsoft.com/library/periodic/period97/html/crypto.htm

5. *Programmation*

[1] Dr. GUI on components, COM, and ATL :
http://msdn.microsoft.com/developer/news/drgui/020298.htm
[2] ActiveX Controls Overview
http://msdn.microsoft.com/library/backgrnd/html/msdn_actxcont.htm
[3] the Visual Programmer
http://msdn.microsoft.com/library/periodic/period97/f1/d2/s245b2.htm
[4] Howto : write C DLLs and Call Them from Visual Basic
Mk :@MSITStore :E:\MSDN\kb.chm ::/Source/vbapps/q106553.htm
[5] Visual C++ Programmer's Guide :
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/devprods/vs6/vc++/vcedit/vcstartpage.htm
[6] A Cryptographic Filter Box Class in Visual Basic
Mk :@MSITStore :E:\MSDN\techart.chm ::/html/msdn/vb_crypto.htm
[7] Security in JDK 1.1.
http://manipulitze.msh.unicaen.fr/Java/doc/tutorial/security1.1/api/index.html
[8] Microsoft Internet Exporer 3.0 Signature and Certificate Interface
http://premium.microsoft.com/msdn/library/backgrnd/html/msdn_iesignwp.htm
[9] How to Remove a Personal Certificate from Internet Explorer
http://support.microsoft.com/support/kb/articles/q185/0/59.asp
[10] Com Techniques by Panther Software
http://www.microwoft.com/workshop/components/com/comatl.asp
[11] Adding a ATL Class
http://www.msdn.microsoft/library/devprods/vs6/vc++/.../_core_adding_an_atl_class.htm
[12] Dr. GUI Does Templates, Dr. GUI Online
http://www.msdn.microsoft/library/welcome/dsmsdn/msdn_090798.htm
[13] The Active Template Library Makes Building Compact COM Objects a Joy
http://www.microsoft.com/msj/0697/atl.htm
[14] Développer une application IIS avec Visual Basic 6 : Partie 1 (introduction)
http://www.microsoft.com/france/msdn/articles/deviis6/default.htm

6. *Smart Cards*

[1] Gemplus lance Gemsafe la solution carte à puce à sécuriser l'accès aux services réseaux
http://www.gemplus.fr/about/pressroom/press/it/gemsafe_fr.htm
[2] Smart Cards, White Paper
http://www.microsoft.com/smartcard/security/tech/smartcards/scardwp.asp
[3] Smart Cards Overview
http://msdn.microsoft.com/isapi/msdnlib.idc?theURL=/library/sdkdoc/scard/scovr_9ws7.htm

7. *Sécurité informatique*

[1] Kaufman, Perlman, and Speciner, *Network Security :Private Communication in a Public World*, Prentice Hall, Upper Saddle River, NJ 07458,1995
[2] Anup K. Ghosh, *E-Commerce Security*, Wiley Computer Publishing, 1998
[3] Andrew Tanenbaum, *Réseaux*, 3ème Ed., Prentice Hall & InterEditions, 1997
[4] Jesse Liberty & J. Mark Hord, *Le programmeur C++*, Simon & Schuster Macmillan, 1996
[5] J. Hubin, *La sécurité informatique :Notes du cours « Sécurité et fiabilité des systèmes informatiques » du Prof. J. Ramaekers, FUNDP, Namur, 1993*
[6] Glossary of Internet Terms Copyright © Matisse Enzer
http://www.matisse.net/files/glossary.html
[7] Data Encryption Techniques
http://www.catalog.com/sft/encrypt.html

8. *Application PBFlow-FUNDP*

[1] Didier Rossetto, *Traitements administratifs d'un dossier « Permis d"urbanisme » article 108*, PBFlow-FUNDP,1998
[2] Didier Rossetto, *WDFB : Un « Workgroup Data Flow Broker » générique*, PBFlow-FUNDP,1998
[3] Frédéric Taes, *TECHNO-WDFB: Introduction aux technologies utilisées dans WDFB*, PBFlow-FUNDP,1998
[4] Bernard Ramaekers, *Sécurisation de PBFlow* ,PBFlow-FUNDP, 1999